Master Data Science
Institut Polytechnique de Paris

Reinforcement Learning

Rapport du Projet

Billiot Bastien
Demay Ulysse
Ilbert Romain

E. Le Pennec

Mars 2021

1 Introduction

Le travail effectué pour ce projet est réalisé à partir de l'article *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model* publié en 2020 et écrit par Julian Schrittwieser, Ioannis Antonoglou, et al.

Dans cet article, les auteurs présentent un nouvel algorithme d'apprentissage par renforcement, *MuZero*, ainsi que ses performances. La motivation principale de cet algorithme et de cet article est de parvenir à atteindre les performances des méthodes à l'état de l'art dans les deux principales catégories de l'apprentissage par renforcement : *model-free* et *model-based*.

L'objectif de tous les algorithmes d'apprentissage par renforcement est d'optimiser le *Markov Decision Process* spécifique à la tâche considérée. Le *model-free Reinforcement Learning* tente d'optimiser une politique directement ou d'apprendre la fonction de valeur sans aucune information sur la dynamique ou la structure de récompense de l'environnement, ce qui représente le problème à résoudre (d'où le nom). A l'inverse, le *model-based Reinforcement Learning* a accès à un modèle de l'environnement et l'utilise dans le processus d'apprentissage pour optimiser la politique. Ces deux méthodes s'adressent traditionnellement à deux types de problèmes qu'utilisent les auteurs pour évaluer et comparer les performances de leur algorithme.

Les domaines et problèmes qui demandent une prévision sophistiquée comme les jeux de stratégies, d'échecs, de shogi et de Go ont vu les méthodes *model-based* émerger comme des solutions à l'état de l'art. Cependant, ces méthodes ne sont pas efficaces lorsqu'il s'agit de problèmes visuellement plus riches comme les jeux Atari 2600. Ce sont alors les méthodes *model-free* qui s'avèrent plus efficaces.

1.1 Méthodes *Model-based* vs.*Model-free*

Les méthodes *model-based* construisent un modèle de l'environnement, modélisé sous la forme d'un processus de décision de Markov qui a deux principaux éléments : un modèle de transition d'état et un modèle de récompense. Une fois que le modèle est construit, ont peut appliquer un algorithme de planification comme le *Monte-Carlo Tree Search*. Un arbre de décision est créé de manière itérative en simulant une série limitée de jeu. Lorsqu'une feuille de l'arbre est atteinte, les informations relatives aux états visités sont rétropropagées dans l'arbre en fonction de la récompense obtenue. Ensuite, une action est choisie. Celle-ci mène à l'état suivant qui rapporte la récompense la plus élevée. A l'inverse, les méthodes *model-free* n'utilisent pas de modélisation de l'environnement mais déterminent les fonctions de valeur directement à partir des interactions avec l'environnement.

2 Présentation de *MuZero*

Motivation

La méthode de l'algorithme présenté et développé par les auteurs de l'article s'inscrit comme une méthode *model-based* qui permet d'obtenir des performances de pointe dans les problèmes classiques de cette partie de l'apprentissage par renforcement : les échecs, le jeu de Go et le shogi ; tout en obtenant également d'excellentes performances sur le problème Atari 2600 traditionnellement obtenues par des méthodes *model-free*. L'algorithme *MuZero* utilise trois composants : la représentation, la dynamique et la prédiction avec pour objectif de prédire les trois quantités futures que sont la politique, la fonction de valeur et la récompense immédiate, importante pour la planification.

[Code] Présentation du code fourni

Dans les parties **[Code]** de ce rapport, nous proposons des références au notebook détaillé fourni avec ce document. Cela permettra au lecteur de faire le lien entre les éléments théoriques issus de l'article et de notre compréhension et les éléments implémentés. Dans le notebook, nous proposons une implémentation fonctionnelle de l'algorithme *MuZero*. Au vu de la complexité de cet algorithme nous avons choisi de reprendre un code déjà implémenté par Johan Gras. Le code est disponible à l'adresse `https://github.com/johan-gras/MuZero` et expliqué tout au long du notebook, commenté et structuré dans des parties correspondant à des concepts introduits dans le rapport.

Dans l'implémentation proposée, *MuZero* n'est pas entraîné sur les jeux de Go, shogi, les échecs ou Atari. En effet, il fonctionne sur le jeu du CartPole, un jeu en 1D avec un faible nombre d'actions possibles et au fonctionnement assez simple. Ce choix à plusieurs raisons. De plus amples explications sur les choix de cette implémentation par rapport à celles de l'article ou des versions plus sophistiquées de *MuZero* sont disponibles en introduction du notebook. Les principales sont que le jeu du CartPole est adapté à l'implémentation de Johan Gras. Par rapport aux jeux présentés dans l'article, il représente une charge computationnelle raisonnable au vu des machines disponibles. Enfin, par rapport à d'autres implémentations plus générales mais plus complexes (dont une est citée dans le notebook) nous voulions également pouvoir expliquer au mieux le code implémenté et en dérouler les étapes de manière similaire à l'article.

A la fin de ce notebook nous commentons les résultats obtenus pour un entraînement sur le CartPole. Nous proposerons donc dans ce rapport d'établir un lien entre les différentes parties de l'article et le code.

[Code] Introduction au jeu

Tout d'abord, le code introduit une première section **Notion de Jeu**. Cette partie implémente des classes majoritairement abstraites qui permettent d'introduire les concepts des algorithmes de reinforcement learning dans le cadre d'un jeu. On retrouve ainsi par exemple une classe `Action()` dont l'objectif est de représenter le concept de réalisation d'un évènement dans un jeu. Nous retrouvons aussi l'introduction de la notion de joueur. De plus, une classe `AbstractGame()` permet de structurer le tour d'un jeu (notamment avec les valeurs reward, actions et les élements de calcul des targets) : une instance de cette classe représente en quelque sorte une photographie du jeu et des valeurs de ses quantités utiles à un tour donné. Ces classes sont ensuite héritées lors de la définition de jeux précis. Par exemple, dans le cas du CartPole, l'environnement du jeu (chargé à partir de la bibliothèque `Gym`) contient l'ensemble des actions possibles dans le jeu. En héritant de la classe `AbstractGame`, la classe `CartPole` permet donc de préciser les éléments d'un tour spécifiques au jeu du CartPole et dont la structure est assurée par la classe abstraite. A partir de l'action sélectionnée, un nouveau tour peut donc être joué.

2.1 La planification

A chaque nouvelle étape temporelle t, le modèle prédit trois quantités futures pour chacune des K étapes hypothétiques considérées (i.e. les K étapes temporelles futures après le temps t) : la politique, la fonction valeur et la récompense immédiate. Nous allons voir cependant que l'algorithme va en réalité calculer 5 valeurs. Pour ce faire, le modèle utilise trois fonctions : la représentation, la dynamique et la prédiction. Les observations passées $o_1, o_2, ...o_t$ et les actions futures $a_{t+1}, a_{t+2}, ...a_{t+K}$ sont également nécessaires. L'état racine s_t^0 est d'abord initialisé à partir des observations passées ainsi que la fonction de représentation, tel que en notant θ les paramètres du modèle :

$$s_t^0 = h_\theta(o_1, o_2, ...o_t)$$

Puis à chaque étape hypothétique, le modèle utilise une fonction de dynamique pour calculer une récompense immédiate r_t^k et l'état s_t^k à partir de l'état précédent et l'action choisie, tel que :

$$r_t^k, s_t^k = g_\theta(s_t^{k-1}, a^k)$$

Une des spécificités de l'algorithme *MuZero* est que l'état s_t^k n'a aucune signification concrète dans l'environnement. C'est un état caché dont le seul objectif n'est pas d'établir un lien avec l'environnement réel mais de prédire les quantités objectifs avec précision.

Enfin, à l'aide de l'état s_t^k, le modèle va prédire, grâce à la fonction de prédiction, la fonction politique et valeur tel que :

$$p_t^k, v_t^k = f_\theta(s_t^k)$$

[Code] Définitions des réseaux de planification

Dans l'implémentation on retrouve la seconde partie **Réseaux** qui contient l'architecture du réseau de l'algorithme adapté au jeu sélectionné et les éléments entraînables. Cette section permet de construire cet algorithme en se basant, comme dans la première section, sur des classes abstraites ensuite héritées par le réseau spécifique au jeu choisi. C'est dans cette partie que vont donc être introduites, sous forme de réseaux, les fonctions décrites ci-dessus. En effet, ces fonctions permettent au temps t, de passer d'une étape hypothétique k à la suivante et de calculer les quantités clés. Elles sont donc nécessaires pour définir l'architecture. Contrairement à l'article, le code n'introduit pas exactement les fonctions de dynamique et de prédiction. Il construit 5 réseaux : un pour la fonction de représentation, un pour prédire un état caché à partir du précédent et de l'action choisie, un pour la value, un pour la récompense immédiate et un pour la politique . L'algorithme va utiliser ces réseaux dans deux classes abstraites (qui ne sont pas spécifiques au jeu choisi).

La première, `InitialModel()`, permet de construire le modèle de l'étape hypothétique initiale et va donc utiliser le `representation_network` qui permet de calculer s^0 (correspond à la fonction de représentation). Elle va également utiliser les deux réseaux issus de la fonction de prédiction : un réseau value et un réseaux politique afin de prédire ces quantités relatives à cette étape initiale.

De même la seconde, `RecurrentModel()`, construira l'étape hypothétique à l'état caché k. Pour ce faire, elle n'utilise plus le réseau de représentation. Elle va concaténer l'état caché de la couche précédente s^{k-1} avec l'action a^k puis passer cet argument dans le réseau dynamique inspiré de la fonction de dynamique et qui permet d'obtenir s^k. L'argument sera aussi utilisé pour déduire la seconde quantité de la fonction dynamique : la reward r^k dans un second réseau. Enfin, comme dans l'état initial, les deux réseaux valeur et politique issus de la fonction de prédiction permettront de déduire les deux quantités à partir du nouvel état caché. Nous avons donc 5 réseaux issus des trois fonctions.

Enfin, ces deux modèles sont assemblés dans l'architecture implémentée `BaseNetwork()` qui les met en oeuvre à l'aide d'autres classes et fonctions abstraites décrites plus précisément dans le code. On retrouve essentiellement les méthodes abstraites de la classe `BaseNetwork()` qui permettent de conditionner, transformer la value, la reward ou les états cachés avec les actions ainsi que l'héritage des classes abstraites `NetworkOutput()` et `AbstractNetwork()` qui permettent de structurer le réseau et ses sorties.

En outre, dans le cadre d'un jeu comme le CartPole, une classe spécifique au jeu `CartPoleNetwork()` hérite de la classe `BaseNetwork()` et va permettre de définir les éléments du réseau propres au jeu. Le résultat est donc

un réseau dont la structure est adaptée à *MuZero* et spécifique au jeu. Il est important de noter que le jeu CartPole étant en 1D, les 5 réseaux présentés plus haut sont des Multi-Layer Perceptrons (et non des réseaux convolutionnels qui sont plus adaptés aux jeux 2D). Dans cette classe nous retrouvons donc l'ensemble des paramètres propres au CartPole, les réseaux et les fonctions abstraites de la classe `BaseNetwork()` particularisées pour le jeu. A ce stade nous avons donc implémenté la notion et l'environnement de jeu ainsi que les réseaux essentiels pour la planification lors de la recherche par *Monte Carlo Tree Search (MCTS)*.

Ces réseaux et plus généralement la phase de planification que nous venons de présenter est résumée dans la figure suivante, reprise de la page 3 de l'article qui illustre l'utilisation de ces fonctions déterminées par les réseaux et les quantités qu'elles permettent d'obtenir.

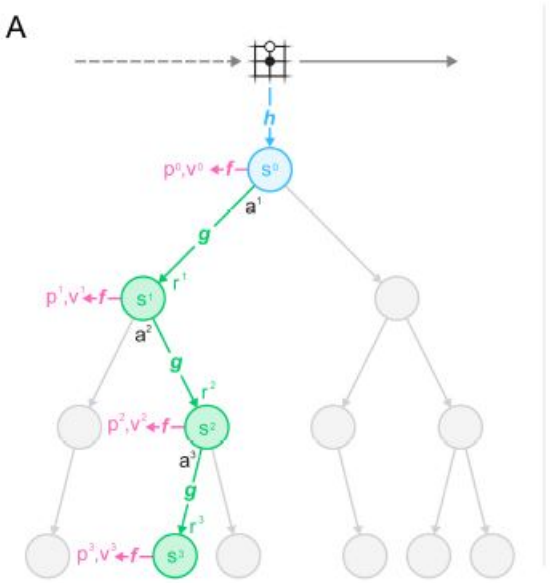

Figure 1 – Illustration de la plannification

2.2 Recherche et interaction avec l'environnement

L'algorithme *MuZero* effectue une recherche arborescente de Monte Carlo (MCTS). C'est un algorithme qui explore l'arbre des possibles. Cet algorithme de recherche conserve en mémoire l'ensemble des arbres déjà explorés. Dans notre cas, à chaque étape de temps t nous effectuons N simulations (i.e. N constructions d'arbre). L'état du jeu au moment présent t permettra d'obtenir la racine de chaque arbre et une feuille correspondra à la configuration d'une étape hypothétique (et éventuellement une configuration finale s'il existe une trajectoire menant à cette configuration en moins de K étapes). Cet algorithme d'arborescence s'effectue en quatre phases :

- Sélection : Depuis la base de notre arbre, nous allons, par un compromis entre exploitation et exploration, choisir les noeuds enfants
- Expansion : Si la feuille n'est pas finale, nous créons un enfant puis nous en choisissons un
- Simulation : Nous simulons l'exécution d'une partie depuis cet enfant jusqu'à la feuille finale
- Rétropropagation : Nous utilisons le résultat de la partie afin de le rétropropager en partant du noeud enfant jusqu'à la racine de l'arbre

Cette arborescence est donc un mécanisme de recherche qui consiste à exécuter un certain nombre de simulations et à construire un arbre de recherche à l'aide de ces résultats. Chaque noeud $n(s)$ de l'arbre de recherche construit au temps t représente un état s déjà observé au cours de la simulation. La phase la plus complexe, parmi celles présentées ci-dessus, est la sélection. En effet, une politique avide pourrait mener à selectionner des actions parmi un petit ensemble, en évitant d'autres actions après seulement quelques mauvais résultats. Cela correspond au compromis exploitation/exploration mentionné plus haut. Pour cette raison, l'article utilise une upper confidence bound en donnant un bonus qui représente l'incertitude face à ce compromis. Ce bonus permettra donc à l'algorithme d'aller plus loin dans cette recherche. La politique de l'arbre sélectionne les actions qui maximisent une fonction de perte que nous définirons ultérieurement.

L'algorithme de recherche arborescente de Monte Carlo permet d'obtenir une politique π_t, qui permet de déterminer une action a_{t+1}. Cette dernière est alors transmise à l'environnement qui générera la nouvelle observation o_{t+1} (qui servira ensuite à la recherche MCTS de l'étape suivante) et la récompense u_{t+1}.

[Code] Monte Carlo Tree Search

La partie **Configuration pour l'algorithme** qui vient avant l'implémentation des MCTS permet de préparer l'implémentation de l'excécution de l'algorithme en introduisant les hyperparamètres nécessaires. On retrouve par exemple le jeu joué, le nombre de boucles d'entraînement, le nombre de fois où l'on va passer à travers un arbre de Monte Carlo, etc. En résumé, on configure et introduit les hyperparamètres pour les différentes phases. Les détails sont donnés dans le code.

Comme présenté ci-dessus, l'algorithme *MuZero* planifie grâce à une recherche MCTS à l'aide des 3 fonctions : représentation, dynamique et prédiction ou des 5 quantités du code dont les réseaux ont été définis. Cette recherche permettra notamment de sélectionner l'action a_{t+1} réalisée dans le jeu entre les temps t et $t+1$. La partie **Monte Carlo Tree Search (MCTS)** se charge d'effectuer cela dans notre code.

- **Selection:** Les premières fonctions reprennent des éléments efficaces de *AlphaZero* pour implémenter la sélection des noeuds enfants à chaque étape. Nous retrouvons donc la fonction `select_child()` qui se charge de la sélection du noeud enfant à visiter par maximisation du score UCB. Le score UCB, dont la formule est en Annexe B de l'article, est également implémenté. Par la suite, nous sortons de cette fonction avec le noeud enfant et l'action qui lui correspond.
- **Expansion d'un noeud:** La phase d'expansion utilise les réseaux des quantités à prédire présentés dans la section précédente et calculera ainsi la valeur de l'état caché s_k, la reward r_k, et la politique p_k. On retrouve ces éléments dans la fonction `expand_node()`. Cette dernière se termine en déterminant les noeuds enfants avec la fonction politique du noeud actuel et les différentes actions possibles.
- **Rétropropagation:** Elle est implémentée dans la fonction `backpropagate()` où les précisions nécessaires sont données dans le code.

Enfin, la fonction `run_mcts()` va structurer la recherche MCTS pour chaque simulation de l'arbre en utilisant les fonctions précédemment définies. La fonction prend en argument la configuration de l'algorithme qui permet de définir le nombre de recherches d'arbres et le noeud racine obtenu grâce à l'état de l'environnement actuel, les actions précédentes et `BaseNetwork()` qui permet de transmettre les réseaux pour obtenir les quantités intéressantes comme décrit ci-dessus et dans la section précédente. A chaque recherche d'arbre, nous avons donc la phase de sélection avec la fonction `select_child()`, la récupération des réseaux, la phase d'expansion de chaque noeud avec

la fonction `expand_node()` et la phase de rétropropagation. Une fois la recherche par MCTS terminée, la fonction `select_action()` permet de sélectionner l'action réelle à réaliser en choisissant à chaque étape de l'arbre le noeud enfant vers lequel nous souhaitons nous diriger, soit par une fonction softmax (pour l'entraînement) soit en choisissant le chemin le plus emprunté lors des simulations (pour l'évaluation). Cette action a_{t+1} est alors transmise à l'environnement. Concrètement, elle est ajoutée à une trajectoire qui contient les différentes actions et un replay buffer conserve les différentes trajectoires.

Pour illustrer cette recherche d'arbre et l'interaction avec l'environnement nous reprenons la figure de l'article en page 3.

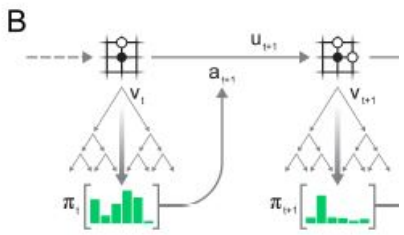

Figure 2 – Illustration de la recherche et action transmise à l'environnement

2.3 L'entraînement du réseau

Pour l'entraînement, le réseau du paragraphe précédent est utilisé sur K étapes hypothétiques et aligné sur les séquences échantillonnées des trajectoires générées par l'arbre de recherche de Monte-Carlo. Ces séquences sont sélectionnées à partir d'un état d'une partie sauvegardée dans un replay buffer, puis déroulées sur K étapes à partir de cet état. Le replay buffer est donc un élément qui nous sert à stocker les données des différentes parties. En résumé, *MuZero* joue des milliers de parties contre lui-même, les enregistre dans un replay buffer et s'entraîne ensuite sur les données de ces parties. De ce point de vue, il n'est pas différent d'*AlphaZero*. L'idée est donc d'échantillonner une trajectoire depuis ce replay buffer. Les observations passées $o_1, ...o_t$ permettent d'obtenir l'état caché initial s^0 grâce à la fonction de représentation. Le modèle est ensuite déroulé pour les K étapes et à chacune d'entre elle, la fonction de prédiction donne la politique p^k et la value v^k à partir de la valeur s^k. Enfin, la fonction dynamique reçoit la valeur de l'état s^k et l'action réelle obtenue par MCTS a_{t+k+1}. On obtient alors par cette fonction la reward r^{k+1} et la valeur de l'état suivant s^{k+1}. Une visualisation de ces étapes essentielles à l'entraînement est présentée dans la figure ci-dessous, extraite de la page 3 de l'article.

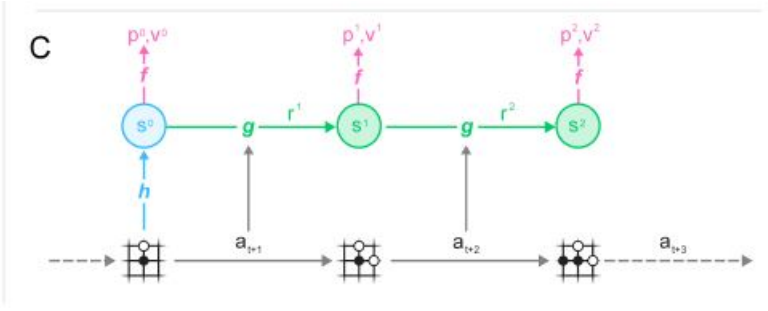

Figure 3 – Illustration des étapes en prévision de l'entraînement

L'entraînement et la détermination des paramètres du modèle sont alors réalisés grâce à une phase de rétropropagation à travers le temps. Ces paramètres sont donc déterminés conjointement dans cette phase d'entraînement. La fonction de perte au temps t, utilisée pour l'entraînement, repose sur trois composantes qui découlent des trois quantités que le modèle prédit à chaque étape k : la politique, la valeur et la récompense. L'objectif est donc, pour chacune des quantités, de minimiser l'erreur entre les quantités prédites respectivement notées p_t^k, v_t^k, r_t^k et les quantités cibles respectivement $\pi_{t+k}, z_{t+k}, u_{t+k}$. Pour chaque quantité l'erreur est évaluée avec une fonction de perte qui lui est propre. Ainsi l'erreur totale à l'étape t, régularisée est la suivante :

$$l_t(\theta) = \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, p_t^k) + c||\theta||^2$$

avec $z_{t+k} = u_{t+1} + \gamma u_{t+2} + ... + \gamma^{n-1} u_{t+n} \gamma^{n-1} u_{t+n} + \gamma^n \nu_{t+n}$

[Code] Entraînement du réseau

Dans notre notebook, cette partie correspond à la section **Entraînement** qui contient trois blocs principaux : le ReplayBuffer dont le rôle a été mentionné ci-dessus, le bloc SelfPlay qui permet de générer les successions d'observations et d'actions d'une partie (donc plus globalement simuler les parties) et enfin le coeur de l'entraînement qui va entraîner les paramètres des réseaux (les paramètres des 5 réseaux) par rétropropagation à partir de la loss définie plus haut. Dans la partie **Notion de jeu**, présentée en début de rapport, se trouve une autre fonction essentielle pour l'entraînement et notamment pour la loss, c'est la fonction `make_target()`.

La classe ReplayBuffer dispose de plusieurs outils pour les gestion des parties stockées. Le premier est simplement de sauvegarder l'ensemble des observations et actions d'une partie grâce à la fonction `save_game()`. La seconde est de renvoyer, parmi un ensemble aléatoire de parties stockées, une "position" (i.e. l'observation du jeu à un tour choisi aléatoirement) aléatoirement choisie de chaque partie. La fonction `sample_games()` sélectionne aléatoirement les parties du ReplayBuffer et la fonction `sample_position()` sélectionne, elle, aléatoirement un tour de la partie. La fonction `sample_batch()` va, une fois que ces éléments sont selectionnés, construire les batchs à renvoyer pour l'entrainement. Un batch est construit à partir d'une liste de tuples, chacun constitué de trois éléments :

- `g.make_image(i)` : l'observation (de la partie) à la position choisie
- `g.history[i:i + num_unroll_steps]` : une liste des prochaines actions effectuées après la position choisie (si elles existent)
- `g.make_target(i, num_unroll_steps, td_steps, g.to_play())` : une liste des cibles qui seront utilisées pour former les réseaux de neurones. Il s'agit d'une liste de tuples : target_value, target_reward et target_policy
- Ce batch initial est ensuite traité pour fournir des informations qui découlent directement de ces trois éléments (précisé dans le code)

Nous entrons ensuite dans la partie Self-play de l'entraînement qui va permettre de jouer les parties. Pour cela nous allons notamment réutiliser des fonctions de la partie **Monte Carlo Tree Search**. Ainsi, la fonction `play_game()` débute une partie. Puis, tant que la fin de la partie ou le nombre maximal d'actions n'est pas atteint, elle part de la dernière observation de la partie en court, utilise la fonction `run_mcts()` pour réaliser la recherche MCTS puis `select_action()` qui permet d'obtenir à l'issu de la recherche MCTS l'action réellement entreprise. Cette fonction `play_game()` joue ainsi successivement les actions sélectionnées et renvoie l'intégralité de la partie. La fonction

`run_selfplay()` va ainsi, dans une boucle d'entraînement, jouer le nombre de parties choisi (paramètre *nb_episodes* dans la configuration de l'algorithme) avec la fonction précédente et les sauvegarder dans le ReplayBuffer. La fonction `run_eval()` permet quant à elle de faire jouer à notre algorithme les parties en mode évaluation. Des précisions sont disponibles dans le notebook.

Avant de décrire la manière dont la fonction de perte est traitée dans le code nous allons présenter la fonction `make_target()`, essentielle pour l'entraînement. Cette fonction utilise des méthodes issues du TD-learning pour calculer les value, reward et politique cibles de chaque état pour des "positions" allant de state_index à state_index + num_unroll_steps. Pour le calcul de la value cible, l'idée principale est que nous pouvons mettre à jour la value d'un état en utilisant la value actualisée estimée d'une position td_steps dans un futur proche à laquelle nous ajoutons les rewards actualisées jusqu'à ce point, plutôt que de simplement utiliser le total des rewards actualisées accumulées à la fin de l'épisode. Le bootstrap_index est l'indice de la position td_steps dans le futur que nous utiliserons pour estimer les véritables rewards futures. Le calcul des rewards et politiques cibles est plus aisé et consiste à utiliser la reward actualisée au point considéré et le nombre de visites des fils d'un noeud donné par la recherche MCTS pour la politique cible.

Nous avons donc des fonctions nous permettant de jouer et stocker nos parties ainsi que des value, reward et politique cibles. Grâce aux parties à disposition, la partie Training de la section **Entraînement** va nous permettre d'entraîner les paramètres des réeaux que nous avons construits précédemment. Dans un premier temps, une fonction de loss spécialisée (basée sur la softmax cross-entropy) est définie pour la value. Ensuite, le coeur de l'entraînement est la définition de la fonction `update_weights()`. Cette dernière introduit et calcule d'abord la loss définie plus haut à partir des batchs et des quantités cibles (voir code pour plus de précisions sur le calcul de cette loss et les étapes dans cette fonction). Ensuite la fonction `update_weights()` optimise par descente de gradient stochastique (méthode choisie dans la classe `MuZeroConfig`) les variables des réseaux à partir de cette loss. Ces variables mises à jour sont celles renvoyées par la fonction `cb_get_variables()` de `BaseNetwork()`, qui ne sont rien d'autres que les poids des 5 réseaux décrits auparavant. Pour finir, la fonction `train_network()` sélectionne, via le `ReplayBuffer()`, les batchs à utiliser pour l'entraînement du modèle puis utilise la fonction `update_weights()` pour l'entraînement avant de sauvegarder les réseaux ainsi entrainés (cette dernière fonction construit l'architecture de l'entraînement).

2.4 Le réseau

2.4.1 L'entrée du réseau

Entrée du réseau pour les jeux de l'article

L'entrée du réseau neuronal est un ensemble d'images $N \times N \times (MT + L)$ qui représente des observations passées d'une partie en utilisant une concaténation de T ensembles de M plans de taille N × N pour le go, le shogi et les échecs. Chaque ensemble de plans représente la position du plateau aux temps $t - T + 1, ..., t$ soit à T étapes ($T \geq 1$) et est mis à zéro si $T \leq 1$. Le plateau est orienté selon la perspective du joueur actuel. Les plans M sont composés de plans binaires indiquant la présence des pièces du joueur, avec un plan pour chaque type de pièce, et un second ensemble de plans indiquant la présence des pièces de l'adversaire. Pour le shogi, il existe des plans supplémentaires indiquant le nombre de prisonniers capturés de chaque type. Enfin, pour Atari, l'entrée de la fonction de représentation inclut les 32 dernières images RGB de résolution 96×96 ainsi que les 32 dernières actions qui ont conduit à chacune de ces images. Chaque image est encodée avec 3 plans, un par couleur.

[Code] Entrée du réseau pour CartPole

Comme décrit dans la partie précédente, le réseau reçoit en entrée pour chaque "position" de chaque jeu composant le batch l'observation du jeu au tour donné, les quantités cibles ainsi que l'historique des actions effectuées. Ici nous allons détailler l'observation du jeu au tour donné par "position". Dans les jeux implémentés celle-ci est donnée par l'environnement `Gym` et mise à jour au cours de la partie selon les actions effectuées.

Pour le jeu du CartPole, l'observation est un vecteur de dimension 4 dont les dimensions sont respectivement : la position du cart, la vitesse du cart, l'angle de la tige et la vitesse de rotation de la tige. Ainsi le réseau global prend en entrée un vecteur de taille 4 via la fonction `initial_inference()` qui permet ensuite de calculer toutes les quantités nécessaires à passer dans les 5 réseaux et définies plus tôt.

2.4.2 L'architecture du réseau

Architecture du réseau pour les jeux de l'article

Nous avons vu dans la section précédente que l'entrée du réseau correspond à une succession d'images. Si la dimension des données en entrée d'un réseau de neurones complètement connecté est importante, par exemple dans le cas d'une image (une image de 28×28 pixels correspondra à un vecteur d'entrée de dimensions 784), le nombre de paramètres θ du réseau sera tellement important que les calculs liés à la phase d'apprentissage seront trop coûteux en temps. Pour répondre à la problématique de la reconnaissance d'image, Yann Lecun a mis au point dans les années 90, en s'inspirant du fonctionnement du cortex visuel des animaux, les réseaux de neurones convolutifs.
L'architecture d'*AlphaZero* consiste en des couches convolutionnelles ainsi que des blocks résiduels qui vont permettre d'extraire l'information du jeu en préservant la résolution mais en réduisant le nombre de plans, suivies d'un réseau entièrement connecté afin de réaliser une tâche de classification. Ce réseau renvoie ensuite une politique qui correspond à un vecteur de probabilité sur toutes les actions possibles. Plus spécifiquement, le réseau est composé comme suit :

- L'entrée décrite dans le paragraphe précédent
- Un convolutional layer composé de 256 filtres de taille 3×3 suivis d'une batch normalization et d'une fonction d'activation ReLu
- 20 blocks résiduels
- Une value head et une policy head

Un bloc résiduel est composé de :

- L'entrée du bloc qui correspond à la sortie de la couche précédente
- 256 filtres convolutionnels de taille 3×3 suivis d'une batch normalization et d'une fonction d'activation ReLu
- 256 filtres convolutionnels de taille de kernel 3×3 suivis d'une batch normalization
- Une skip connection qui nous a permis de garder en mémoire les informations de l'entrée du bloc qui sont ici ressorties
- Une fonction d'activation ReLu

Ainsi, 20 blocs comme celui-ci s'enchaînent en cascade (16 pour le jeu atari dont la structure est un peu différente du réseau *AlphaZero* ci-dessus) jusqu'à une value head et une policy head qui vont représenter respectivement la sortie du réseau pour la valeur et pour la politique.
Pour la valeur, la structure du réseau est la suivante :

- L'entrée qui correspond à la sortie de la cascade des blocks résiduels
- Un filtre convolutionnel de taille 1×1 suivi d'une batch normalization et d'une fonction d'activation ReLu
- Une couche de 256 neurones entièrement connectée suivie d'une fonction d'activation ReLu
- Une couche de neurones entièrement connectée suivie d'une fonction d'activation tanh

La sortie est donc un scalaire qui prend des valeurs entre -1 et 1 et correspond à la valeur du jeu pour le joueur.
Pour la politique, la structure du réseau est la suivante :

- L'entrée qui correspond à la sortie de la cascade des blocks résiduels
- Deux filtres convolutionnels de taille 1×1 suivi d'une batch normalization et d'une fonction d'activation ReLu
- Une couche de neurones entièrement connectée
- Une fonction d'activation Softmax qui donne la probabilité sur chacune des $19 \times 19 + 1$ actions possibles

La sortie est donc un un vecteur de taille 362 composé des probabilités de chacune des 362 actions.

[Code]Architecture du réseau pour CartPole

Nous pouvons retrouver l'implémentation de l'architecture de nos 5 réseaux dans la classe `CartPoleNetwork()`. En effet, comme présentée plus haut, cette dernière précise les éléments, dont les réseaux des quantités, spécifiques au jeu du CartPole. Comme mentionné précédemment, le jeu du CartPole étant en 1D, les réseaux de neurones utilisés sont des Multi-Layer Perceptrons dont voici les spécificités.

L'architecture du réseau de représentation est la suivante :

- Les entrées sont les observations qui ont été présentées dans la partie code du paragraphe précédent (entrée du réseau)
- Une couche Dense (Fully-Connected) composé de 64 neurones cachés (précisé dans les paramètres de `CartPoleNetwork()` et d'une fonction d'activation ReLu
- Une seconde couche Fully-Connected avec une sortie de dimension 4 (dimension de l'état de l'étape initial) et une fonction d'activation tanh.
- Pour rappel, la sortie est s^0

La structure des 4 autres réseaux (valeur, politique, dynamique et reward) est la même à savoir deux couches Fully-Connected. Cependant les entrées, fonctions d'activation et sorties peuvent etre amenées à changer.

Ainsi nous avons pour la valeur :

- Entrée : s^k (sortie du réseau de représentation à l'étape initiale ou du réseau dynamique sinon) de dimension 4
- Nombre de neurones cachés (première couche) : 64
- Activation de la première couche : ReLu
- Dimension de la sortie : 24
- Pour rappel, la sortie est v^k

Pour la politique :

- Entrée : s^k (sortie du réseau de représentation à l'étape initiale ou du réseau dynamique sinon) de dimension 4
- Nombre de neurones cachés (première couche) : 64

- Activation de la première couche : ReLu
- Dimension de la sortie : 2
- Pour rappel, la sortie est p^k

Pour la dynamique :

- Entrée : Concaténation de l'état caché précédent s^{k-1} et l'action a^k de dimension respective 4 et 2.
- Nombre de neurones cachés (première couche) : 64
- Activation de la première couche : ReLu
- Dimension de la sortie : 4
- Activation de la seconde couche : tanh
- Pour rappel, la sortie est s^k

Pour la reward :

- Entrée : Concaténation de l'état caché précédent s^{k-1} et l'action a^k de dimension respective 4 et 2:
- Nombre de neurones cachés (première couche) : 16
- Activation de la première couche : ReLu
- Dimension de la sortie : 1
- Pour rappel, la sortie est r^k

3 Éléments de comparaison avec *AlphaZero*

3.1 Les règles du jeu

AlphaZero était jusque lors désigné comme l'algorithme permettant d'obtenir de bons résultats assez rapidement sans nécessiter de connaissances préalables contrairement champions humains d'échecs par exemple. *MuZero* va encore plus loin. Non seulement il ne nécessite pas de connaissances préalables, mais il n'est pas non plus nécessaire de lui apprendre les règles du jeu. Prenons par exemple le jeu d'échecs, *AlphaZero* apprend à jouer et gagner par lui-même mais on lui fournit les règles du jeu à savoir : comment les différentes pièces peuvent se déplacer, les coups licites et également quand un coup est un échec et math ou un pat (égalité), ce qui met fin à la partie. À l'inverse, *MuZero* n'a pas besoin des règles générales du jeu. On ne lui fournit que les coups licites dans la position actuelle et quand une partie est terminée (échec et math ou pat). *MuZero* va notamment "appréhender" les règles générales dans les états cachés s^k décrits précedemment. Ces derniers n'ont cependant pas de connexion réelle avec l'environnement.

3.2 L'entraînement

MuZero joue des milliers de parties contre lui-même, les enregistre dans un buffer puis entraîne et met à jour les paramètres des réseaux à partir de données qu'il tire de ce buffer. De ce point de vue il est similaire à *AlphaZero*. La différence principale s'effectue au niveau du réseau.

3.3 Le réseau

AlphaZero et *MuZero* utilisent tous deux la technique de Monte Carlo Tree Search (MCTS) pour sélectionner le meilleur coup suivant. L'idée est que pour sélectionner le meilleur coup suivant, il est logique de jouer des scénarios futurs probables à partir de la position actuelle, d'évaluer leur valeur en utilisant un réseau de neurones et de choisir l'action qui maximise la valeur future attendue.

Cependant, *MuZero* ne connaît pas les règles du jeu et n'a aucune idée de la manière dont une action donnée affectera l'état du jeu. Il ne peut donc pas imaginer les scénarios futurs dans l'arbre de recherche. *MuZero* apprend à jouer le jeu en créant un modèle dynamique abstrait de l'environnement et en l'optimisant dans le cadre de ce modèle.

Alors qu'*AlphaZero* n'a qu'un seul réseau de neurones (prédiction), *MuZero* en a besoin de trois (prédiction, dynamique, représentation). Le réseau neuronal servant à la prédiction pour *AlphaZero* consiste à prédire la politique p et la valeur v d'un état de jeu donné. La politique correspond à une distribution de probabilité sur tous les coups et la valeur n'est qu'un nombre unique qui permet d'estimer les récompenses futures. Cette prédiction est faite chaque fois que l'arbre touche un nœud de feuille non exploré.

MuZero dispose également d'un réseau neuronal de prédiction , mais l'état de jeu sur lequel il fonctionne est une représentation cachée qu'il apprend à faire évoluer grâce à un réseau neuronal dynamique. Ce réseau dynamique prend l'état caché actuel s et l'action choisie a et produit une récompense r et un nouvel état. *AlphaZero* passe d'un état à l'autre dans l'arbre de recherche à l'aide de l'environnement à l'inverse de *MuZero* qui doit construire son propre modèle dynamique. Enfin, pour passer de l'état de jeu observé actuel à la représentation initiale, *MuZero* utilise un troisième réseau de neurones servant à la représentation.

Finalement, en l'absence des règles actuelles des échecs, *MuZero* crée un nouveau jeu dans son esprit qu'il peut contrôler et l'utilise pour planifier l'avenir. Les trois réseaux (prédiction, dynamique et représentation) sont optimisés de façon conjointe.

3.4 Les résultats des auteurs

Les auteurs de *MuZero* ont fait tourné leur algorithme sur les jeux d'échecs, de Go, de Shogi et sur le jeu Atari. L'entraînement se déroule sur un nombre d'étapes $K = 5$ sur un million de mini-batches de taille 2048 pour les échecs, le Go et le Shogi, et de taille 1024 pour Atari. L'algorithme utilise, durant l'entraînement et l'évaluation, 800 simulations à chaque recherche pour les jeux de plateau et 50 pour le jeu Atari. Les résultats donnés dans l'article sont les suivants :

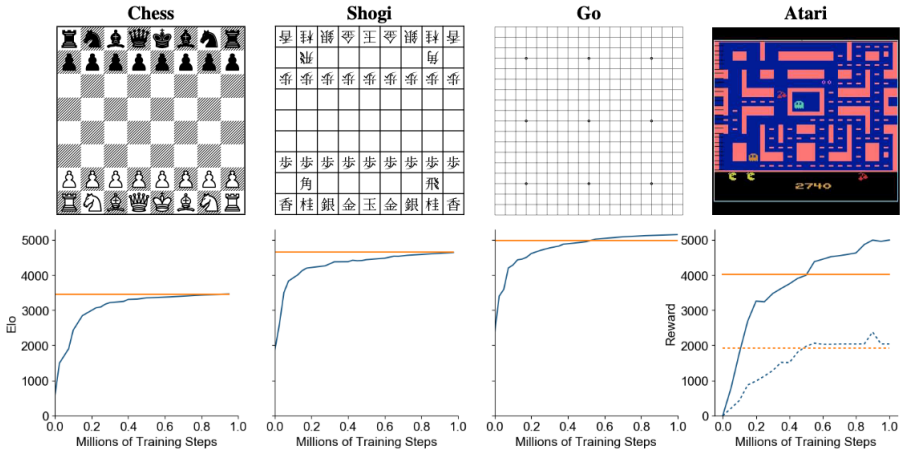

Figure 4 – Résultats de l'algorithme *MuZero* à travers divers jeux

L'axe des abscisses représente le nombre de training steps en millions. Celui des ordonnées représente le ELO pour les jeux de plateau et le reward pour le jeu Atari. La courbe bleue représente le résultat de l'algorithme *MuZero* à travers les itérations, et la orange le résultat de l'algorithme *AlphaZero* à travers les itérations pour les jeux de plateau. Pour le jeu Atari, la ligne pleine et celle en pointillées représente respectivement la moyenne et la médiane des scores de l'état de l'art R2D2 qui représente une approche de type model-free.

Ainsi, nous remarquons que pour les jeux de plateau l'algorithme *MuZero* fait aussi bien qu' *AlphaZero* dans le cadre des échecs et du Shogi et dépasse même sa performance pour le jeu de Go. Dans le cadre du jeu Atari, *MuZero* surperforme légérement l'état de l'art pour la médiane, et largement dans le cadre de la moyenne. On voit clairement que la motivation principale de l'article est atteinte: les deux auteurs ont réussis à développer un modèle d'apprentissage par renforcement *model-based* permettant d'atteindre le niveau des algorithmes à l'état de l'art à la fois dans des jeux de plateaux complexes (domaines de *AlphaZero*, donc déjà du *model-based*) mais aussi dans des domaines visuellement riches comme les jeux Atari dont l'état de l'art était traditionellement réservé aux algorithmes *model-free*. Ce succès repose essentiellement sur la combinaison de trois réseaux ou fonctions à la place d'un qui permettent la représentation cachée.

4 Valeur ajoutée par le groupe

4.1 Les éléments de travail exclusif

En raison de la complexité de l'algorithme et du niveau de détail du pseudo-code fourni avec l'article, nous avons choisi d'utiliser l'implémentation de *MuZero* réalisée par Johan Gras. Nous avons transformé le code sous forme de fichiers .py en un code adapté à un notebook, ceci afin de pouvoir intégrer une structure plus facilement implémentable et compréhensible. Ainsi, notre travail a une vocation explicative du code et de l'article afin de comprendre pleinement cet algorithme *MuZero*.

Notre implémentation apporte également quelques ajouts par rapport à celle reprise du dépôt Git. En effet, nous avons implémenté par nous-même une partie permettant de réaliser une sauvegarde locale et de recharger les différents réseaux obtenus à la suite de l'entraînement de l'algorithme. Ainsi, vous trouverez dans l'archive du projet un dos-

sier contenant un réseau entraîné pour le CartPole pour 12 boucles d'entraînement (ce qui représente quelques heures d'entraînement). Nous avons également implémenté une partie résultat contenant un graphe présentant les moyennes des scores d'entraînement et d'évaluation pour 3 entraînements (de 12 boucles chacun) sur le jeu CartPole ainsi que la descritption et l'interprétation de celui-ci.

4.2 Les résultats obtenus

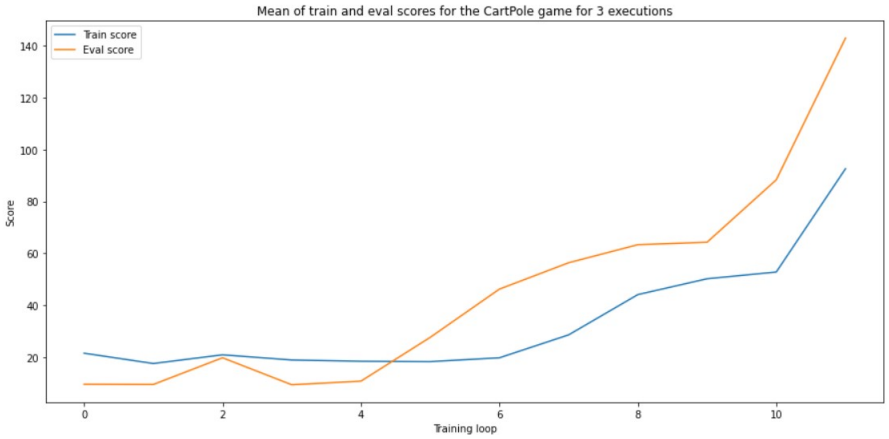

Figure 5 – Résultats de notre implémentation de *MuZero* sur le jeu du CartPole

Voici les résultats obtenus lors de l'exécution de *MuZero* par notre notebook sur le jeu du CartPole. Cette figure est la moyenne de trois entraînement durant 12 boucles d'entraînement chacun afin d'obtenir une courbe donnant l'allure des scores de train et d'évaluation pendant l'entraînement. Les scores observables sur le graphe sont la moyenne des récompenses obtenues pour chaque partie de CartPole en mode train ou en mode évaluation.

Notre notebook décrit et interprète ce graphe en détails. Globalement le score augmente, et ce de plus en plus fort au fur et à mesure des boucles d'entraînement, cela montre bien que notre algorithme apprend et fonctionne donc pour le CartPole. Le phénomène le plus notable visible est que le score d'évaluation est inférieur au score de train au début de l'entraînement alors que le réseau utilisé pour les parties d'évaluation est postérieur à celui des parties d'entraînement. Cela est principalement dû à l'exploration présente en mode train qui permet de ne pas choisir de politique trop cupide et donc d'obtenir un score trop faible.

5 Conclusion

Dans ce rapport, nous avons présenté l'algorithme *MuZero* à partir de l'article *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model*. Nous avons fait une présentation de cet algorithme, avons étudié ses principales composantes telles que les arbres de recherche de Monte-Carlo ou encore l'architecture du réseau utilisé avec les différentes fonctions à apprendre, ainsi que les éléments qui permettent de le distinguer de l'algorithme *AlphaZero*. Nous avons également fourni un notebook détaillant et structurant une implémentation de *MuZero*. D'autres parts, nous avons fait un lien tout au long de ce rapport entre notre développement théorique et explicatif de l'article et notre implémentation. L'algorithme a pu être entraîné sur un jeu en 1D et les résultats moyennés de plusieurs entraînement affichés et commentés. Notre implémentation apporte quelques ajouts au code initial comme la possibilité de sauvegarder et recharger

un réseau entraîné. Pour terminer, les auteurs proposent dans cette article un algorithme qui va plus loin que précédemment avec un modèle à l'état de l'art sur des problèmes aux solutions traditionnellement séparées entre *model-free* et *model-based*. Ils y parviennent en proposant l'algorithme *MuZero* qui, notamment, n'a pas besoin des règles précises du jeux.

6 Bibliographie

[1] *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model.* Julian Schrittwieser, Ioannis Antonoglou, et al. arXiv 2019

[2] *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis. 2017

[3] https://github.com/johan-gras/MuZero. Implémenation de *MuZero* par Johan Gras qui a servi de base.

[4] https://github.com/werner-duvaud/muzero-general. Implémentation générale de *MuZero* qui n'a pas servi dans le travail mais a été consultée.