RL for RTS

BEROUKHIM Keyvan
NGUYEN Laura

1) Introduction

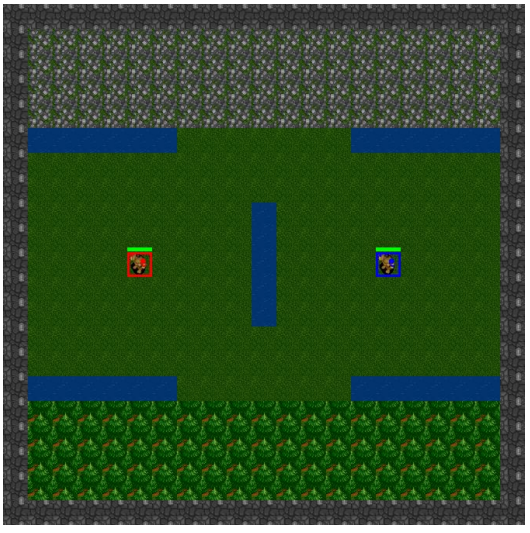
a) Type d'algorithme utilisé

De manière générale, pour créer un agent intelligent, l'idée la plus intuitive est de baser les choix d'action de l'agent sur sa compréhension du monde, c'est à dire d'utiliser un algorithme "model-based".

Cependant, si les algorithmes model-based ne sont pas les plus répandus en reinforcement learning c'est qu'ils font face à plusieurs difficultés que les autres algorithmes ne rencontrent pas. Afin d'avoir une meilleure idée des difficultés rencontrées par les algorithmes model-based, nous allons ici essayer de concevoir et d'utiliser un algorithme **model-based** simple.

b) Environnement

Le but du projet est d'appliquer des algorithmes de Reinforcement Learning pour un jeu de stratégie en temps réel simplifié. L'espace des actions est discret et de petite taille : nous considérons les 4 actions de déplacement basiques ainsi que l'action 'récolte de ressource'. Durant ce projet, nous avons choisi de nous intéresser uniquement à la **récolte de ressource**. Nous choisissons d'initialiser la carte avec un bâtiment principal déjà construit et nous modifions l'environnement pour sélectionner automatiquement les unités présentes sur la carte.



Carte du jeu. Le jeu ne possédant pas une bonne documentation, la prise en main de l'environnement a été difficile et nous suspectons la présence de quelques bugs.

2) Apprentissage du monde

a) Motivation

Les algorithmes de **RL sont difficiles à débugguer**. L'entraînement d'un agent se fait d'un bloc, il ne se décompose pas en plus petites étapes. Si l'agent ne performe pas bien, il n'y a pas de propriétés que l'agent est censé vérifier et dont on pourrait vérifier la validité.

En entraînant de manière supervisée un **réseau apprenant le fonctionnement du monde**, à la différence des algos de RL classiques on sait ce que le réseau est censé retourner. Cela permet de vérifier la prise en main de l'environnement et d'obtenir des architectures capables de comprendre leurs entrées. L'algorithme hybride model-based Dyna-Q vu en cours est conçu pour des espaces discrets de petite dimension, il n'est pas applicable ici.

b) Architecture générale du réseau

L'apprentissage supervisé des réseaux de neurones fait l'hypothèse que les données qui lui sont présentées sont indépendantes et issues de la même distribution. Afin d'apprendre l'effet de chaque action on apprendre **un réseau indépendant par action** afin que l'apprentissage d'une action n'affecte pas les autres. Chacun des 5 réseaux prend en entrée l'état du jeu et retourne l'état que l'on obtient en effectuant l'action correspondante. Les réseaux prennent en entrée une représentation de la carte concaténée à un vecteur représentant l'état du joueur. Chaque case est représentée par plusieurs "one-hot" vecteurs concaténés, par exemple le type de terrain ("herbe"/"ressource"/"mur") ou le type d'unité.

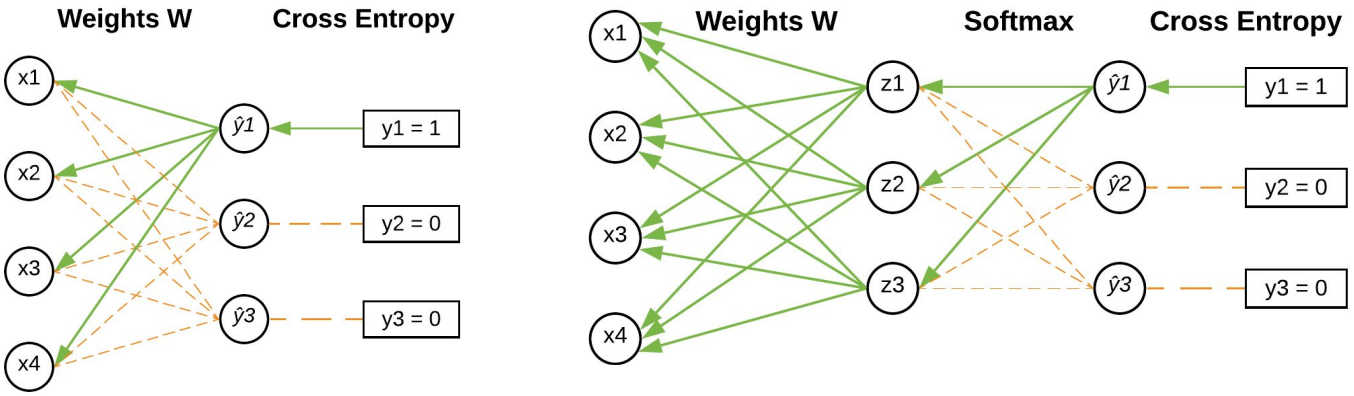
Afin de faciliter l'apprentissage, le réseau prend en entrée une **fenêtre centrée** sur l'unité effectuant l'action. Nous aurions pu utiliser un softmax en sortie des réseaux pour chacun des vecteurs concaténés mais nous décidons de ne pas rajouter de connaissances expertes du monde et de garder cette représentation simple.

Plutôt qu'apprendre à générer la sortie, il est plus simple pour le réseau d'apprendre à générer la **différence entre l'entrée et la sortie**. Cependant, avec un réseau codant une différence, il n'est pas pratique d'utiliser une sigmoïde en sortie du réseau car une valeur de 0 ou 1 en entrée correspond à une valeur infinie avant application de la sigmoïde (on pourrait potentiellement convertir les entrées en ε et 1 - ε avec ε un paramètre du modèle).

c) Fonction de coût

Pour l'apprentissage de la carte, les valeurs à prédire étant dans [0, 1] on commence par utiliser une '**cross-entropy**' comme loss : CE(y || y_pred) = - somme y * log(y_pred). Un premier problème est que sans sigmoïde en sortie du réseau, les valeurs peuvent être négatives auquel cas la cross-entropy n'est pas définie. Une première idée serait de repartir sur l'entraînement d'un réseau ne codant pas la différence.

Une autre solution explorée est de ramener les valeurs prédites dans le bon intervalle en faisant un 'clip' (ou 'clamp') sur la sortie. Afin de permettre une rétropropagation des gradients, nous faisons l'expérience consistant à modifier la fonction 'backward' en retournant un gradient non nul pour les valeurs sortant de l'intervalle.



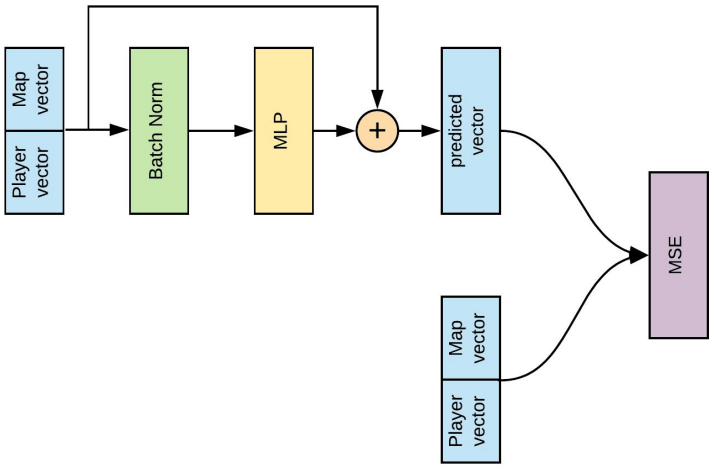
Sans softmax, la plupart des poids ne reçoivent pas de gradient
Avec softmax, tous les poids reçoivent en théorie des gradients non nuls

Après avoir réalisé de nombreuses expériences, nous avons compris que **sans softmax, utiliser une cross-entropie n'est pas souhaitable**. En effet, sans softmax les sorties du réseau associées à un label nul ne reçoivent pas de gradient. Dans notre cas, certaines cases ont tout le temps un label nul et notre réseau ne peut donc pas les apprendre. L'idée d'utiliser la cross-entropy dans l'autre sens n'est pas non plus applicable car on obtiendrait des log(0) pour tous les labels nuls.

Utiliser une **MSE** comme loss résout simplement les problèmes de définition de la loss et de valeur du gradient.

d) Architecture de chaque sous réseau

Un autre problème auquel nous faisons face est la **grande variance dans les entrées du réseau**. En effet, pour pouvoir gérer les actions comme la récolte de ressource (et aussi pour la création d'unités), les réseaux doivent avoir en entrée et en sortie le vecteur d'état du joueur, cependant, alors que la carte est représentée par des valeurs entre 0 et 1, la quantité de ressource peut varier de 0 à 10^5. Sans utiliser de BatchNorm, nous n'avons pas réussi à entraîner les réseaux. Bien que l'utilisation de BatchNorm ne soit pas dommageable dans ce cas d'apprentissage supervisé, l'utilisation de BatchNorm en reinforcement learning peut biaiser le modèle.



Architecture d'un réseau de neurone apprenant l'effet d'une action

De la même manière, nous n'avons pas réussi à bien entraîner nos modèles en utilisant l'**optimiseur** basique "SGD" mais seulement en utilisant "Adam". Mais il se trouve que le momemtum des optimiseurs tels qu'Adam est aussi profitable en reinforcement learning.

e) Utilisation du réseau

Une fois le réseau appris, pour l'utiliser en inférence, les valeurs du monde étant entières, on **arrondit les sorties du réseau vers l'entier le plus proche**. La minimisation de la loss (qui se fait avec des outputs non arrondis) fait tomber le nombre de mauvaises prédictions (la '0/1-loss') à 0 pour la plupart des actions.

2) Agent A* / Best First Search

a) Fonctionnement de l'agent

L'agent possède 3 modules :

- '**A**' : l'ensemble des réseaux apprenant l'effet des **actions** (supervisé)
- '**R**' : un modèle apprenant les **rewards** obtenus (supervisé)
- '**V**' : un réseau de neurone représentant la **valeur** des états (RL)

On commence par effectuer des actions aléatoirement afin de récolter des transitions. Cela permet de commencer à apprendre les modules A et R comme vu précédemment. Les valeurs des états dépendent de la politique, le module V est initialisé en apprenant les gains amortis moyens obtenus à partir de chaque état.

À chaque tour de jeu, en se servant du réseau A, l'agent génère un **graphe d'états** possibles sous la forme d'un arbre ayant pour racine l'état actuel de l'agent. Les noeuds sont évalués par V et par les gains récoltés par R en suivant la trajectoire qui nous a amené de la source vers ce noeud. L'agent explore en priorité les noeuds ayant la plus grande valeur, à la manière d'un algorithme "Best First Search". L'agent explore un nombre de noeuds fixés ou jusqu'à trouver un gain non nul. La valeur des états peut aussi être interprétée comme l'heuristique de l'algorithme A*

Les états visités sont stockés dans une **table de hachage**. Lorsqu'on liste les états accessibles depuis un noeud, si un état a déjà été vu et que les rewards obtenus par cette nouvelle trajectoire ne sont pas meilleurs, nous ne l'ajoutons pas au graphe. Par exemple si l'agent effectue les actions 'droite' puis 'haut' ou 'haut' puis 'droite' les états obtenus sont identiques (si il n'y a pas de mur) et nous n'ajoutons dans ce cas qu'un seul noeud au graphe. Cela permet d'élaguer de nombreuses branches et donc de réduire considérablement l'espace de recherche.

b) Mesure du temps d'exécution

Le temps d'exécution est réparti à peu près équitablement entre :

- l'extraction du vecteur représentant le monde autour de l'agent
- des multiplications matricielles
- la mise à jour de la carte à partir du vecteur retourné par A
- le calcul de la valeur de hash (calculer un hash à partir de la carte entière est très lent, considérer la position des unités et l'état du joueur est beaucoup plus rapide)
- comparaisons de deux noeuds de même hash

c) Expériences

Nous définissons les rewards qui guident l'agent vers la collecte de ressource nécessaire à la création d'une unité : l'agent reçoit un reward quand il récolte des ressources dont le total ne dépasse pas 50. Cela crée une situation où l'agent ne doit pas tout le temps répéter les mêmes actions.

L'agent arrive à récolter les deux types de ressources si il a une **profondeur de recherche suffisante.** Si l'agent ne regarde pas assez loin pour repérer un reward positif, il base ses décisions sur l'heuristique V. Cependant, même en ayant une connaissance parfaite du monde, avec un réseau V entraîné sur des actions aléatoires, suivre l'heuristique peut nous amener vers un **extremum local** d'où aucun reward ne pourra être reçu. En recevant un reward négatif à chaque itération, l'agent va petit à petit baisser la valeur des états de cette zone jusqu'à ce qu'il préfère aller voir ailleurs.

A partir du moment où l'agent parvient à atteindre une ressource, il pourrait y arriver de plus en rapidement en terme d'action et en terme de temps de calcul en mettant à jour son heuristique. Cependant, au vu du temps mis par l'agent pour récolter une fois chacune des ressources, nous ne réalisons pas cette expérience. En effet :

- Pour chaque noeud ouvert, l'agent exécute chacun de ses 6 réseaux (1 pour la valeur de l'état et 5 pour les actions possibles)
- Sans avoir à disposition une heuristique informative, pour trouver une case à distance d, l'agent doit explorer un nombre de noeuds de l'ordre de $4d^2$.
- Si on répète ces calculs à chaque tour de jeu (afin de pouvoir détecter des changement dans l'environnement par exemple) il faut effectuer de l'ordre de 1000 passes forward des réseaux de neurones pour pouvoir atteindre une ressource se trouvant à seulement 5 cases de l'agent.

3) Conclusion

Bien que l'on ne se serve pas des gradients traversant les réseaux apprenant le fonctionnement du monde, l'utilisation de réseaux de neurones dans ce cadre supervisé nous permet d'obtenir un schéma général d'architecture adapté à cette tâche. Sans avoir une loss nous indiquant les performances de notre architecture, nous aurions eu plus de difficulté à comprendre qu'un réseau peu profond peut suffire à partir du moment où la description du monde est aplatie et les entrées normalisées. Nous aurions alors passé beaucoup de temps à entraîner différents agents possédant des architectures neuronales complexes sur un grand nombre de trajectoires.

On a réalisé que l'entraînement des réseaux par renforcement peut être complexe si les données ne sont pas normalisées. Notre heuristique se heurte à ce problème que nous n'avons pas réussi à résoudre.

Malgré quelques optimisations des performances algorithmiques, notre implémentation naïve d'un agent model-based est très coûteuse en temps de calcul aussi bien pour l'entraînement de l'agent que pour sa simple utilisation.

Notre réseau peut s'adapter aux changements d'environnement tels que les actions du joueur adverse. Cependant, il n'est pas conçu pour modéliser des probabilités. Si l'environnement le nécessitait, une solution serait par exemple d'apprendre des GAN pour chaque action, mais cela augmenterait encore plus la complexité de l'agent.

En conclusion, bien que la technique d'exploration d'états de notre approche ressemble à un raisonnement humain, la complexité de calcul très importante de notre agent ne reflète pas l'aisance d'un joueur humain pour réaliser ce genre de tâche.

La complexité de notre méthode permet de mieux comprendre les avantages des algorithmes couramment utilisés tel que l'algorithme "Deep Q-learning" par exemple. Cet algorithme parvient à ramener ce problème complexe à l'entraînement d'un unique réseau de neurone simple, qui étant donné un état, modélise l'espérance de gain de chaque action.