CentraleSupélec

Evaluation and improvement of a deep reinforcement learning model for the planning of condition-based maintenance operations in a large-scale industrial system

Pôle projet IA – 2022/2023

Groupe 03

Etienne CAMUS

Julien CARDINAL

Camille LANÇON

Théophile ROUSSELLE

Paul TABBARA

Encadrants

Jean-Philippe POLI

Wassila OUERDANE

SNCF orange™ EDF

I-Introduction

1) Présentation du client

Notre client est la Chaire RRSC (Risques et Résiliences de Systèmes Complexes) et notre interlocuteur privilégié est un doctorant membre de cette chaire : Matthieu Roux. Cette chaire, qui reprend celle sur les sciences des systèmes et les défis énergétiques (SSEC) soutenue par EDF pendant 9 ans, a pour objectif l'élargissement du champ d'applications au-delà de la production d'énergie. Ses principaux partenaires sont EDF, la SNCF et Orange qui bénéficient ainsi de connaissances et données mutualisées. Ainsi, un intérêt de notre projet sera l'adaptabilité de son rendu aux différents domaines d'activités de ces partenaires.

L'équipe de la chaire travaille majoritairement avec des processus stochastiques et des approches data, d'où le lien avec l'Intelligence Artificielle. D'ailleurs, nous utiliserons ici des processus stochastiques tels que les chaînes de Markov.

2) Présentation du sujet et attentes du client

Les systèmes industriels tels qu'un banc d'éoliennes offshore, une flotte de trains ou encore des centres de données sont des systèmes complexes constitués d'éléments se dégradant au cours du temps. Le but de ce projet est de trouver une politique de maintenance optimale pour un réseau constitué de n'importe lequel de ces éléments. Pour ce faire, nous allons utiliser des outils de l'IA tels que le RL (Reinforcement Learning) ou encore le BDQN (Branching Dueling Q-network). Le principe sera le suivant : étant donné un certain nombre d'items (nous considérerons le cas des éoliennes dans la suite de ce rapport) qui suivent une certaine loi d'usure, nous allons devoir optimiser leur politique de maintenance en prenant potentiellement en compte de nombreux facteurs, tels que les différents types de maintenances (préventives et correctives), le temps de travail à notre disposition, un bridage des items selon leur niveau d'usure, la production des éoliennes, etc. Il faudra donc trouver un compromis entre un modèle réaliste mais assez simplifié pour pouvoir le mettre en œuvre.

Le client attend de notre groupe de se familiariser avec les différents problèmes de maintenance et avec les principes du RL et du BDQN, leurs forces et faiblesses, leur implémentation etc. Cela nous permettra de construire et d'implémenter un premier modèle non-deep et simplifié, qui constituera notre MVP, puis de tuner ses hyperparamètres. Enfin, il

sera possible d’améliorer le modèle en le complexifiant ou en le rendant plus réaliste par exemple. Ainsi, il sera intéressant d’essayer d’élargir notre travail au deep pour réussir à gérer des environnements plus vastes et donc plus proches de la réalité.

II- Etat de l'art

1) Les algorithmes de RL

Les premiers algorithmes sur lesquels nous nous sommes penchés sont les algorithmes non profonds (ou non deep) de Reinforcement Learning (RL). L'objectif d'un algorrithme de RL est d'entraîner un agent, qui est un système observant l'état d'un environnement et retournant une action pouvant altérer l'état de ce dernier. Nous nous sommes intéressés en particulier à la méthode de Monte Carlo et la différence temporelle.

- Monte Carlo apprend sur des exécutions entières de l'agent sur l'environnement, appelés samples. Il nécessite que l'environnement ait une fin dans le temps et fonctionne particulièrement bien pour des environnements avec des rewards très sparse, peu au cours du temps avec une reward importante à la fin. Notre problème de maintenance étant sur un temps infini, on a mis cet algorithme.
- La différence temporelle permet d'apprendre tout en parcourant l'environnement. L'algorithme le plus connu est celui de Q-learning qui utilise SARSA avec une politique comportementale en epsilon greedy et une politique d'apprentissage greedy. C'est cet algorithme qu'on a utilisé.

La politique comportementale est la politique epsilon greedy qui certifie une bonne exploration, avec une probabilité epsilon de faire une action au hasard. Cette politique n'est pas parfaite pour trouver le chemin optimal car nous aimerions qu'après avoir eu une bonne exploration, nous puissions bien exploiter le meilleur chemin et arrêter d'explorer. Ainsi, nous faisons décroitre epsilon de manière exponentielle : il est multiplié par 0.99 tous les 500 steps.

De cette manière, l'agent apprend la politique optimale tant qu'epsilon est assez grand et l'exploite ensuite. Cette notion exploration/exploitation est cruciale pour tout agent de RL.

Le facteur de dévaluation gamma est lui aussi très important pour l'apprentissage. Il intervient dans la formule d'estimation du Gain : $G = \sum_{i=0}^{+\infty} \gamma^i R_{t+i}$

Dans notre cas, il faut prendre en compte les gains moyens à très long terme, jusqu'à 5000 jours, c'est pour cela que nous prenons γ très proche de 1 : 0.9999 par exemple donne pour i = 5000 on a $\gamma^i \approx 0.6$ qui donne l'importance des reward 13 ans plus tard (si 1 step = 1 jour)

Le facteur gamma est aussi responsable de la convergence des Q-Values car avec un gamma très grand, on approxime la somme qu'avec un grand nombre de mise à jour de la Q-Value.

Enfin le facteur gamma est aussi responsable du trade-off biais variance, avec un biais élevé pour gamma petit et une variance faible mais un biais faible et une variance élevée pour un gamma grand (proche de 1). Ici, en choisissant un gamma élevé, on se retrouve à devoir lutter contre une variance importante.

2) Le DQN

Nous nous sommes ensuite penchés sur des algorithmes de Deep Q-learning dont l'objectif est de faire apprendre les valeurs d'actions par un réseau de neurones. C'est donc l'implémentation de l'agent qui va être différente, l'environnement reste le même.

Fonctionnement du DQN : Après avoir choisi une action, l'environnement évolue, se retrouvant dans un nouvel état et l'agent récupère les informations observées sur l'environnement (états de dégradation des éoliennes et la reward liée à l'action prise). A partir de l'état des éoliennes, l'agent calcule la sortie du réseau de neurones qui représente les valeurs d'action des actions réalisables à partir de cet état de l'environnement, et l'agent peut également prendre un décision grâce aux valeurs d'action obtenues et à une politique définie. Grâce à la reward obtenue, l'agent peut optimiser son réseau de neurones par rapport à une certaine fonction de perte. La phase d'optimisation ne se fait pas à chaque étape de décision mais plutôt sur un batch d'étapes, c'est dire qu'un batch est constitué de plusieurs étape dans lesquelles l'agent a pris une décision d'action mais sans faire évoluer son réseau de neurone, et à la fin du batch, l'agent optimise le réseau par rapport à toutes les étapes du batch.

Nous avons considéré une politique epsilon-greedy pour la prise de décision. Cela signifie qu'à chaque prise de décision, l'agent a une probabilité epsilon de choisir une action aléatoire et une probabilité 1 – epsilon de suivre une politique greedy, c'est-à-dire choisir l'action maximisant les valeurs d'action.

Pour la fonction de perte, nous avons utilisé une première fonction décrite par PyTorch [1] :

On définit un objectif pour la valeur d'action d'un état donné :

$$Q^{s}_{target} = r + \gamma \cdot \max_{a} Q^{s,a}$$

$$avec\ \gamma = 0.99$$

L'idée ici est d'estimer la valeur réelle de Q^s par méthode de bootstrapping :

$$Q^{s} = \sum_{t \geq t_0}^{+\infty} \gamma^{t-t_0} R_t = R_{t_0} + \gamma Q^{s+1}$$

Ensuite, on définit la fonction de perte sur une étape comme la différence entre la valeur d'action de l'action que l'on a effectuée à l'état s, et la valeur d'action objectif :

$$\delta^{s,a} = Q^{s,a} - Q^{s}_{target}$$

Pour la fonction de perte sur tout un batch, on prend la moyenne de toutes les fonctions de perte pour toutes les étapes du batch (B est la taille de batch) :

$$L = \frac{1}{B} \cdot \sum \delta^{s,a} \qquad \text{(1) Fonction de perte 1}$$

Nous avons également considéré une deuxième fonction de perte tirés d'un TP sur le DQN [3] :

$$L = -\frac{1}{B}(\sum reward) \cdot (\sum Q^{s,a}) \qquad \text{(2) Fonction de perte 2}$$

L'idée de cette fonction de perte est de maximiser les valeurs d'action si elles sont à l'origine de reward élevées et inversement de minimiser les valeurs d'action à l'origine de reward faibles.

3) Le BDQN

Le DDQN (pour Dueling Deep Q-Network) est une architecture RL évoquée pour la première fois en 2016 dans [4], présentant l'avantage de mieux estimer les valeurs d'état que les architectures de l'ancien état de l'art.

Les BDQNs (Branching Dueling Q-Networks) décrits en [5] donnent une amélioration des DDQN avec une structure d'arbre permettant le partage de la prise de décision et donc une meilleure mise à l'échelle. [6] nous montre une application des BDQN dans la planification de tâches pour des micro grilles de systèmes de stockages d'énergies distribués, pouvant nous conforter de leur utilité dans notre situation, assez similaire.

Cependant, nous n'avons pas eu le temps de mettre en pratique cette littérature nouvelle dans notre projet.

4) Bibliographie

[1] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html]

[2] https://automatants.cs-campus.fr/formations : Slides de la formation "Reinforcement Learning – DQN" de 2021-2022

[3] TP dans le cadre du camp "ML4Good" de EffiSciences : https://www.effisciences.org/

[4] Ziyu Wang et al., Dueling Network Architectures for Deep Reinforcement Learning, 2016

[5] Arash Tavakoli et al., Action Branching Architectures for Deep Reinforcement Learning, 2019

[6] Hang Shuai et al., Branching Dueling Q-Network Based Online Scheduling of a Microgrid With Distributed Energy Storage Systems, IEEE TRANSACTIONS ON , VOL. , NO. , 2021

[7] : Reinforcment Learning Course by David Silver (deepmind) https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ&index=1

III- Travail réalisé

1) Démarrage et répartition des tâches

La durée du projet étant assez court, il était crucial d'avoir une organisation efficace assez vite pour se plonger directement dans l'implémentation. Pour l'organisation technique, nous avons choisi d'utiliser gitlab car nous étions à peu près tous familier avec l'outil.

Le projet étant en 2 parties : la compréhension du problème avec l'implémentation de l'environnement et la création d'agents performant dessus, nous avons décidé de segmenter au maximum le travail de création de l'environnement pour pouvoir travailler en parallèle sur des fonctions différentes. Ainsi, dans un premier temps, Paul s'est chargé des états, Etienne des transitions d'un état à l'autre, Théophile de visualiser au mieux les états et les performances de l'agent, Julien des actions et de la fusion de chacune des parties pour avoir un code fonctionnel et Camille de l'implémentation du premier agent de RL pour tester le bon fonctionnement de l'environnement.

L'environnement étant créé, nous avons par la suite tous essayé de développer nos parties et de rajouter des éléments manquants. Paul a implémenté un agent de référence, Etienne a développé les matrices de transition possible, Julien a simplifié l'environnement pour permettre à l'agent d'être plus performant dessus et Camille s'est chargé d'implémenter un agent deep aussi aidé par Paul.

Au-delà de cette répartition axée sur la performance, nous voulions aussi que tous les membres du groupe aient une bonne compréhension du sujet et des algorithmes utilisés. Ainsi chacun des membres a implémenté un agent de Q-learning sur un environnement simplifié, pour comprendre les le fonctionnement de l'agent.

2) Création de l'environnement

a- Etats, reward et actions

Les principales contraintes dans la création de l'environnement étaient de fournir un comportement compatible avec des algorithmes de RL, assez flexible afin de pouvoir s'adapter aux différentes requêtes de potentiels clients. Pour ce faire, nous avons considéré qu'un environnement était constitué de plusieurs items. Chaque item possède un seuil de panne, un état de santé, une fonction modélisant l'évolution de l'état de santé de l'item (renvoyant un

delta d'état de dégradation) et une fonction de production d'énergie pouvant dépendre de l'état d'usure de l'item. Une fois que l'usure d'un item dépasse le seuil défini, ledit item est considéré hors-service et ne produit plus d'énergie. Cette modélisation permettait de valider une certaine flexibilité de l'environnement : en effet, le seuil d'usure pouvant être entier tout comme réel, et les deltas de dégradation pouvant être entiers comme réels, nous avons une architecture s'adaptant aussi bien au modèle discret qu'au modèle continu. Afin de réduire la taille de l'espace d'observation, nous avons choisi de représenter les états sous forme de liste triée par valeur d'usure : en effet, en supposant les items homogènes dans les paramètres qui les décrivent, seules les usures des items sont pertinentes pour décrire le comportement d'un agent.

Sur chacun de ces items, des actions peuvent être prises parmi : maintenance préventive, maintenance corrective (initialement on avait instauré la possibilité de brider la production d'énergie de l'éolienne, qui aurait pour effet de réduire l'usure mais nous avons décidé de la retirer par souci de complexité). Une maintenance préventive permet de réduire l'état d'usure d'un item en service tandis qu'une maintenance corrective permet de passer d'un état hors-service à un état fonctionnel. Dans la réalité, il existe des contraintes physiques restreignant le nombre de maintenances réalisées par semaine. Dans le code, nous avons choisi d'assigner un poids à chacune de ces actions (0,1 et 0,3 respectivement). L'action est codée comme un tuple dont la première valeur correspond au nombre de maintenances préventives choisies et la deuxième au nombre de maintenances correctives choisies. Une action est alors valable si elle a un poids total inférieur à 1.

Une fois ces actions définies, on leur assigne un coût. Une partie est fixe, liée dans le cas d'un parc éolien offshore à l'affrètement d'un bateau de maintenance par exemple, et une autre variable en fonction des maintenances choisies. Nous avons exprimé le coût de chaque élément en fonction de l'énergie apportée par une éolienne en un jour. Afin d'obtenir des coûts réalistes, nous avons choisi d'assigner des coûts de 54 unités pour les coûts fixes et de maintenance corrective, et des coûts de 18 unités pour les maintenances préventives. Tout ceci nous donne alors une méthode de construction de reward : quotidiennement, on soustrait à l'énergie totale créée par le parc les possibles coûts associés aux maintenances. On verra par la suite qu'on a parfois choisi des reward différentes, en moyennant les reward quotidiennes des n jours suivant la prise de décision par exemple.

b- Modélisation de l'usure

Une fois les différents états, actions et reward qui régissent l'environnement implémentées, il fallait s'occuper de la modélisation des items, et plus particulièrement de leur loi d'usure. Dans un souci de simplicité, nous avons décidé de commencer par une modélisation discrète du niveau d'usure d'une éolienne, et n'avons finalement pas trouvé le temps pour

expérimenter une modélisation continue, jugeant le rapport entre le temps passé à adapter l'environnement et l'intérêt de cette modélisation trop important.

Nous avons ainsi opté pour une modélisation Markovienne de l'usure d'une éolienne : nous considérons un certain nombre d'états de fonctionnement et un certain nombre d'états de panne (si l'on distingue les équipements de l'éolienne par exemple) et modélisons les transitions entre ces différents états par une matrice stochastique. De plus, cette matrice est triangulaire supérieure, car on considère que l'éolienne ne peut pas « améliorer » son état sans intervention, et lorsque l'éolienne se trouve dans un état de panne, elle y reste. Nous avons d'abord essayé de trouver des ressources sur les lois d'usure d'une éolienne pour adopter un modèle le plus réaliste possible et avons expérimenté certains modèles, avant de demander directement à Matthieu Roux qui nous a fourni la matrice suivante pour un environnement à 4 états de fonctionnement et un état de panne, construite sur la base d'observations :

$$\begin{bmatrix} 0.9848 & 0.015 & 0 & 0 & 0.0002 \\ 0 & 0.989 & 0.01 & 0 & 0.001 \\ 0 & 0 & 0.98 & 0.005 & 0.015 \\ 0 & 0 & 0 & 0.94 & 0.06 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 1 : Matrice d'usure stochastique

Ainsi, à partir d'un état de fonctionnement, on a une grande probabilité d'y rester, mais on peut aussi directement tomber en panne ou passer au niveau d'usure suivant. En traçant l'état d'une éolienne suivant ce modèle au cours du temps, on remarque qu'on est quasi-sûrs d'être en panne au bout de 500 jours sans intervention, ce qui ne semble pas aberrant puisqu'une éolienne sur terre a une durée de vie moyenne d'entre 20 et 30 ans, avec de l'entretien.

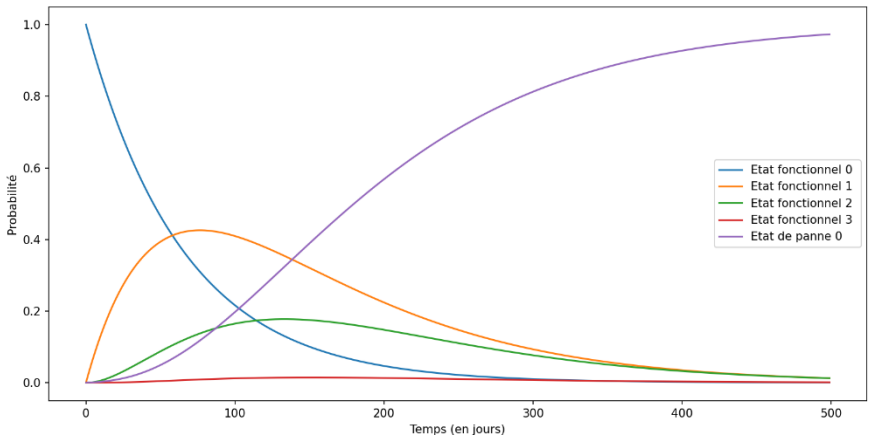

Figure 2 : Répartition des états au cours du temps

3) Implémentation des agents baseline

a- Agent réflexe

Dans un premier temps, nous avons implémenté un agent réflexe comme premier outil de mesure de l'efficacité des algorithmes proposés. L'agent réflexe est un algorithme naïf prenant des décisions dès qu'il le peut. Ici, ce n'est pas réellement le cas mais le principe reste le même : dès qu'au moins un item est en panne, l'agent va déclencher autant de maintenances correctives que possible afin de réparer tous les items. Ensuite, si des items présentent une usure au-dessus d'un certain seuil, l'agent va déclencher autant de maintenances préventives que possible, pour essayer d'améliorer la santé des items au-dessus du seuil. Avec un seuil de déclenchement des actions à 75% du seuil de panne, nos premiers agents sous-performaient par rapport à l'agent réflexe, soulevant ainsi des faiblesses à résoudre.

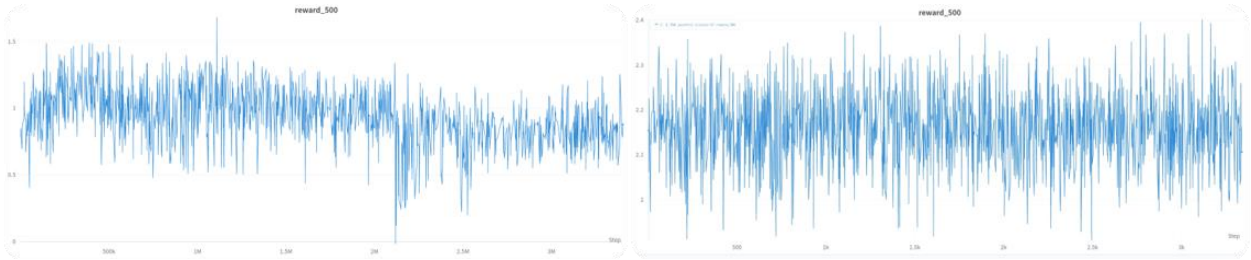

Figure 3:Comparaison de la reward de l'agent fixe (droite, toujours supérieure à 2) et de celle de l'agent de Q-learning présenté à la soutenance intermédiaire (gauche, toujours inférieure à 1,5)

b- Agent solveur

Afin d'évaluer la performance de notre modèle, monsieur Roux nous a suggéré d'utiliser un solveur MDP (Markov Decision Process). Théophile s'est donc attelé à cette tâche, tâche estimée à 12 heures de travail. Nous nous sommes d'abord posé la question du choix du solveur. En effet n'ayant aucune expérience dans ce domaine il fallait d'abord baliser le terrain. Nous avons par exemple envisagé CPLEX, pour sa réputation, cependant l'École ne nous permettait pas d'y avoir accès et ce dernier semblait difficile d'utilisation. Au cours de nos recherches nous avons fini par trouver la bibliothèque MDP toolbox sur python. Il s'agit d'une bibliothèque comportant plusieurs algorithmes de résolution de chaînes de Markov à temps discret, bibliothèque que nous avons choisie car elle était facile à implémenter, et les résultats facilement exploitables.

Nous avons ensuite choisi de faire en sorte que le solveur fonctionne avec comme argument la matrice stochastique A d'écrivant l'usure markovienne d'une seule éolienne et p le nombre d'éoliennes, pour simplement pouvoir voir quelle est la limite de ce dernier et comparer notre travail à ces performances dans plusieurs conditions. Il a donc fallu, à partir des deux arguments choisis, fournir les deux arguments nécessaires aux solveurs qui sont : une matrice Q de dimension (a,s,s) avec a le nombre d'actions possibles et s le nombre d'états et R la matrice de reward de dimension (a,s,s) mais où chaque colonne a une unique valeur, elle est donc isomorphe à un matrice (a,s). La matrice Q est une matrice telle que chaque sous-matrice Q[i] soit une matrice stochastique décrivant la possibilité de passer d'un état à un autre après avoir décidé l'action i. L'une des difficultés était dû à l'un de nos choix permettant de réduire grandement notre espace d'état en considérant que les états : [1,1 ,2], [1,2,1] et [2,1,1], où l'argument n de liste représente l'état d'usure de l'éolienne n, sont équivalents. Ainsi pour un système avec 3 éoliennes et 5 états d'usure possibles nous avions un espace d'état de 35 au lieu de 125. Enfin il a fallu implémenter le délai de x jour entre la prise de décision d'une action et sa mise en œuvre. Pour cela il suffisait de multiplier tous les éléments, sauf Q[0], de la matrice Q par Q[0]^x, où Q[0] est la matrice stochastique correspondant à ne rien faire, raisonnement que l'on appliquera également à R. Notons qu'il est assez simple dans notre cas d'implémenter le délai, car nous avons fait l'hypothèse qu'une fois une décision prise, il est impossible d'en prendre une autre dans le délais de x jours.

Une fois les différents arguments entrés nous devons choisir l'algorithme que mdp.toolbox utilisera pour résoudre le problème. Nous avons choisi l'algorithme de backward induction car c'était celui qui était natif dans la documentation mais nous n'avons pas approfondi en regardant tous les algorithmes de résolutions possibles. Mdp.toolbox nous fournit donc comme solution à notre problème les matrices de valeurs d'états et de politique. Remarquons que pour 20 éoliennes nous avons arrêté l'algorithme au bout d'une vingtaine de minutes car il ne donnait pas de résultat. Il aurait pu être intéressant de le faire tourner sur une machine plus puissante que les nôtres. Voici donc le résultat pour un modèle de 3 éoliennes avec 5 états de vieillissement (4 de fonctionnement et un de panne d'après la présentation de l'environnement) :

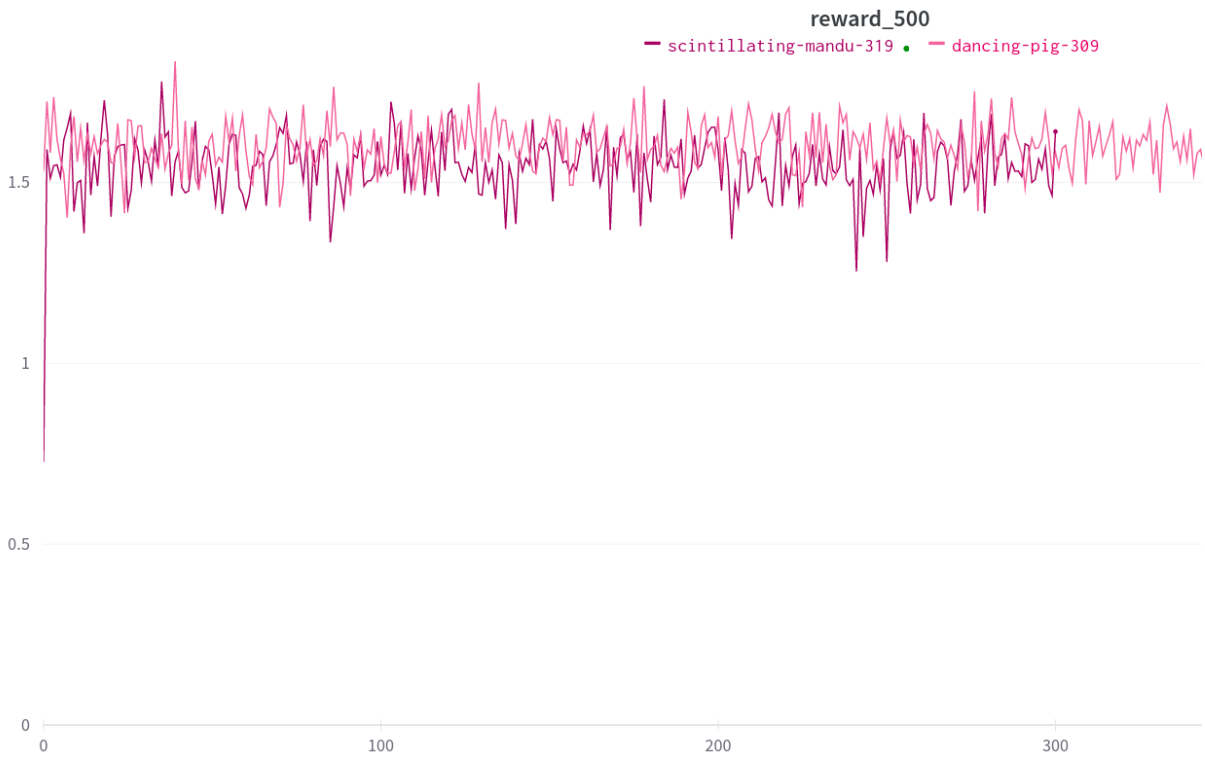

Figure 4 : en rose le solver en mauve notre agent

Les résultats de notre agent sont relativement proches de ceux du solver, ce qui permet d'attester de la qualité du travail réalisé.

4) Implémentation des agents de RL

a- Agent de Q-Learning

Nous avons commencé par l'implémentation d'un agent de Q-Learning sur l'environnement et nous avons directement constaté un problème avec la forme de la courbe de notre apprentissage : avec notre epsilon qui décroit trop vite, on se retrouve coincé à l'état où toutes les éoliennes sont hors service.

Comment expliquer ça ? En fait epsilon permet à l'agent de prendre des réparations de manière régulière. C'est pour cela qu'avec un epsilon pas trop grand, on se retrouve avec une reward moyenne positive. Avec cet epsilon, l'agent apprend à se laisser porter par epsilon et

apprend tout simplement à ne rien faire, c'est pour cela qu'il ne fait rien quand epsilon n'est plus là.

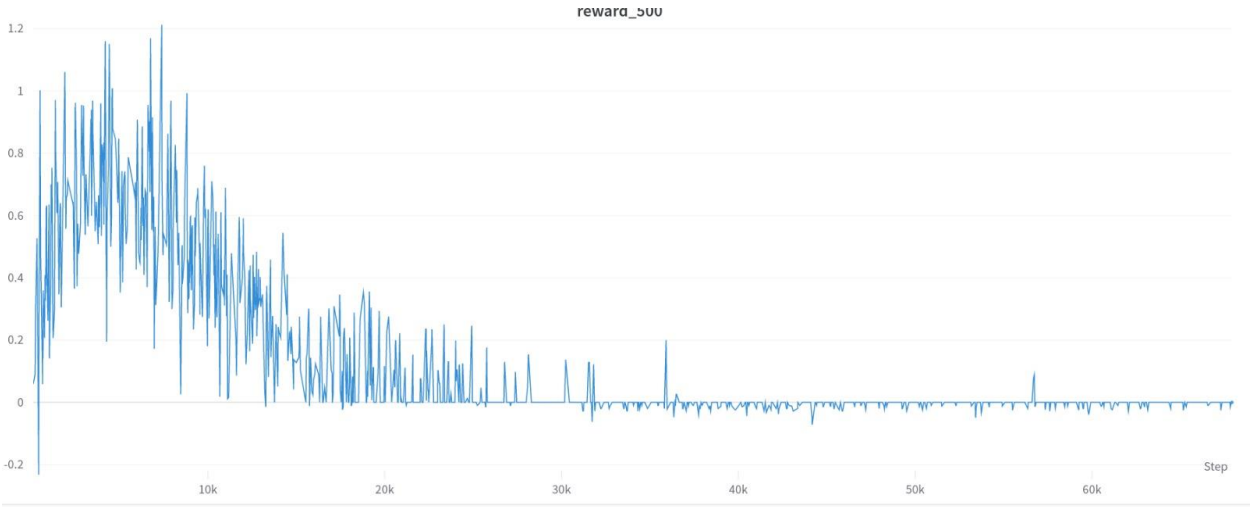

Figure 55 : Agent aidé par epsilon apprend à ne rien faire

Pour améliorer l'environnement et favoriser l'exploration de l'agent, nous avons fait 2 choses :

(1) Nous avons regroupé les actions les actions « ne rien faire » tel que l'on puisse ne prendre des décisions que sur un changement d'état
(2) Nous avons centré la reward pour obtenir une reward moyenne nulle, permettant de faire converger plus facilement les Q-Values

Pour expliquer ces décisions et leur impact, nous pouvons nous pencher sur un exemple d'environnement très simplifié :

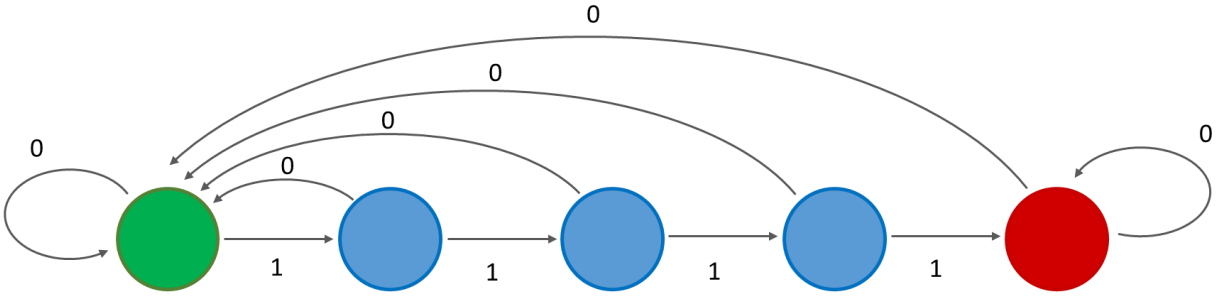

Figure 66 : Graphe d'un environnement simple similaire

Dans cet exemple, sachant qu'on essaie d'optimiser le gain moyen par unité de temps, le chemin le plus optimal est définitivement celui allant du nœud vert au nœud rouge et revenant au nœud vert. On peut appeler ça une boucle qui donne 4/5 de reward par temps.

Dans ce cas-là, avec un epsilon important, disons 0.5, on va rester bien plus dans les premiers nœuds : le premier vert et le bleu suivant, descendre jusqu'au rouge a une probabilité de 0.5^4 = 0.0625 donc avec un epsilon important on a un déséquilibre de l'exploration. Avec epsilon faible au contraire, on descendra jusqu'au bout mais avec une très mauvaise exploration.

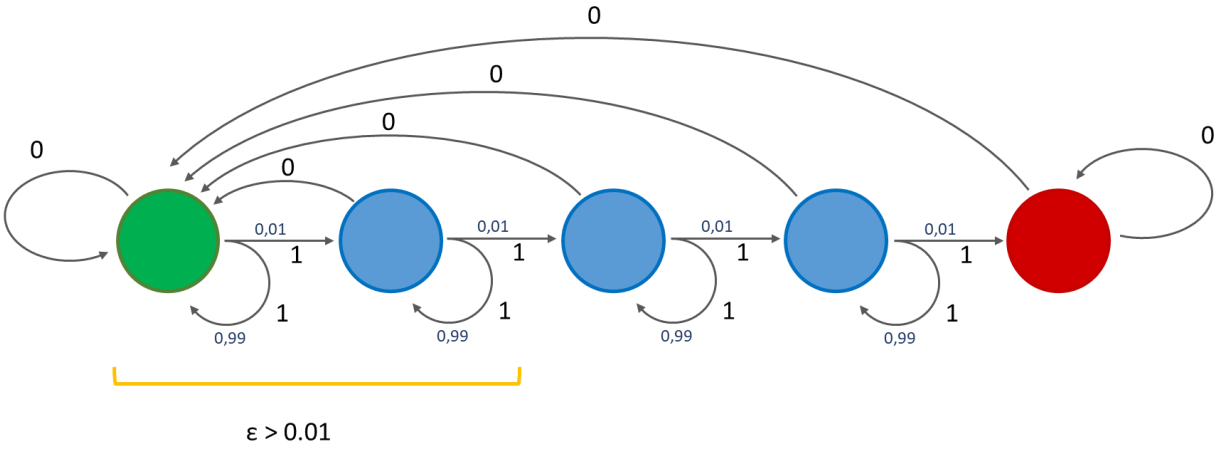

Figure 77 : Graphe stochastique avec possibilité de rester dans les états

Notre environnement sans l'amélioration (1) ressemble plus à la figure ci-dessus, avec des probabilités à chaque fois de rester dans le même état. Dans ce cas-là, si on permet à l'agent de prendre des décisions à chaque t, il faudrait epsilon extrêmement petit pour avoir une chance de visiter les états dégradés (état proche de l'état rouge). On a donc dans ce cas une exploration totalement asymétrique.

Grâce à (1) on peut passer de ce graphe à celui d'au-dessus avec une reward proportionnelle au nombre de fois où les probabilités font qu'il reste dans le même état.

Pour expliquer (2), on peut comparer le premier graphe à celui-ci-dessous :

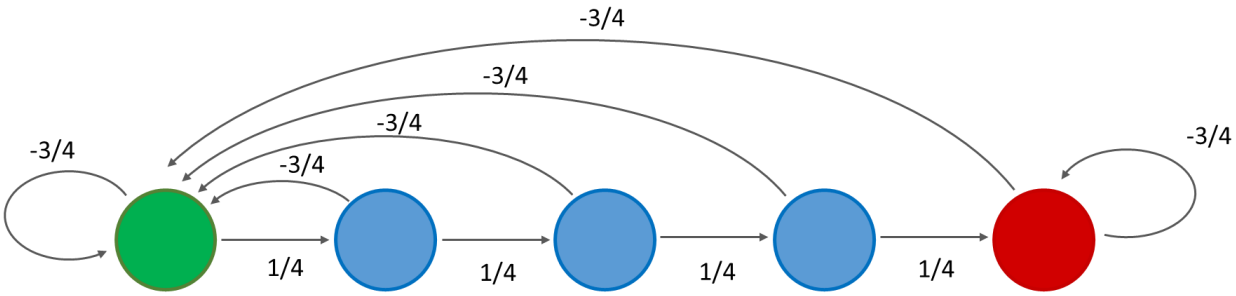

Figure 88 : Graphe avec les rewards centrées

Dans le premier graphe, toutes les boucles donnaient une reward positive (ou nulle), l'agent, aidé par une asymétrie d'exploration a tendance à explorer seulement les état peu dégradés (proche de vert) et la Q-Value de ces états (+actions) est l'espérance du gain sur le long terme va monter en flèche pour ces boucles positives. Ainsi les Q-Values des états peu explorés sera très petite comparé à ceux beaucoup explorés. L'agent va donc choisir de continuer à exploiter les mêmes boucles.

Avec (2), toutes les boucles sont négatives ou nulle sauf une qui gagne en moyenne ¼. Cette fois-ci, les Q-Values des états des boucles proches de l'état vert seront négatives ce qui poussera l'agent à tester les états avec des Q-Values moins négatives, ce qui le forcera à aller jusqu'à l'état rouge et à trouver la meilleure boucle.

On pourrait dans ce cas même avoir un agent qui n'utilise pas epsilon pour l'exploration.

L'exploration est maintenant maitrisée et on trouve la politique optimale assez rapidement pour un petit nombre d'éoliennes, ce qui est montré par la comparaison au solveur.

On peut donc se permettre de recomplexifier l'environnement, en ajoutant par exemple du délai dans l'exécution des actions : en mettant un délai entre la prise de décision et l'exécution de l'action, de 14 jours, on se retrouve avec une différence de reward moyenne, reflétant la perte de réactivité face aux aléas d'usure de l'éolienne :

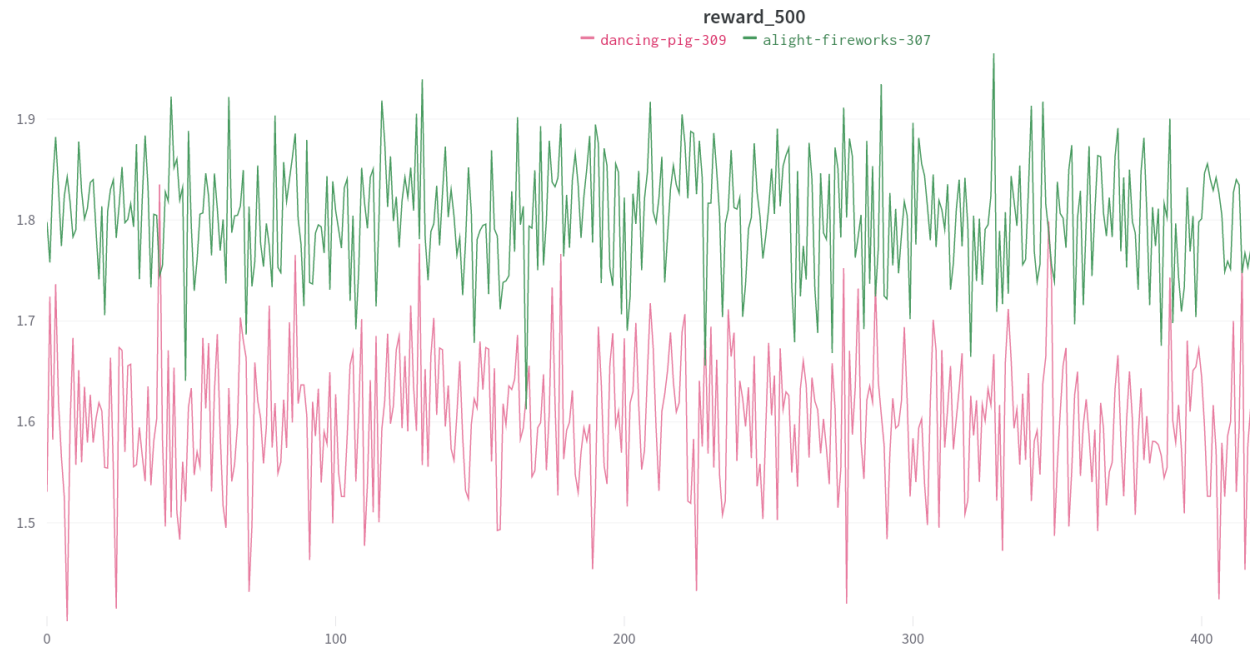

Figure 9 : Comparaison de l'agent avec et sans délai

Avec une production qui diminue de 11.25% entre la courbe verte (sans délais) et la rose (délais de 14j)

On a de plus essayé l'algorithme sur un plus grand nombre d'éoliennes, sachant que le nombre d'états augment de façon quasi-exponentielle. L'algorithme tourne mais l'agent rencontre des difficultés à apprendre face au nombre immense d'états. On peut imaginer qu'avec une décroissance plus lente d'epsilon sur un supercalculateur, on arriverait à le faire tourner.

b- Agent de Deep Q-Learning

Nous avons implémenté un agent de DQN avec un réseau de neurones dense et une politique epsilon-greedy. Nous avons dû décider de la structure de réseau (entrée, sortie, nombre de couches et de neurones), de la forme de la fonction de perte et d'autres hyperparamètres du réseau (fonction d'activation, earning rate, taille de batch).

- Entrée du réseau : un vecteur avec les taux d'usure de toutes les éoliennes. Les taux d'usures sont compris entre 0 et 1 avec 0 quand l'éolienne n'est pas du tout dégradée et 1 quand l'éolienne est hors-service.

  Exemple pour 3 éoliennes : [0; 0.25;1]

- Sortie du réseau : un vecteur des valeurs d'action pour chaque action possible.

Une action est ici un ensemble avec un certain nombre de maintenances préventives et un certain nombre de maintenances correctives.

Exemple d'action: {2: preventive; 1: corrective}

- Politique de décision : une politique epsilon-greedy avec un epsilon qui décroît au cours du temps (on multiplie epsilon par un facteur 0.99 tous les 100 steps.
- Fonctions de perte : nous avons implémenté les fonctions de pertes 1 et 2 expliquées dans l'état de l'art afin de comparer les deux.
- Nombres de couches et de neurones : nous avons implémenté un code qui permet de changer très facilement le nombres de couches et de neurones. Nous avons travaillé avec un nombre de couches allant de 1 à 3 et un nombre de neurones de 32 ou 64.
- Nous avons choisi d'utiliser l'optimiser Adam pour l'entraînement avec un learning rate de 0,01.
- Pour la taille de batch, nous l'avons fait varier entre 5 et 250.
- Nous avons utilisé la fonction d'activation "ReLU" pour les couches internes et nous avons testé plusieurs fonctions d'activation en sortie : "ReLU", "Tanh", "Softmax", "Sigmoid".

Méthodologie : afin de choisir les paramètres, nous avons fait plusieurs entraînements en changeant les paramètres (un seul à la fois) pour tester des valeurs différentes.

Résultats : on obtient principalement 2 résultats différents en fonction des paramètres choisis.

1) On obtient un premier résultat où la reward moyenne est très basse et instable. Ce résultat est obtenu pour la première fonction de perte lorsque le gamma est trop élevé (gamma=0.9999 par exemple).

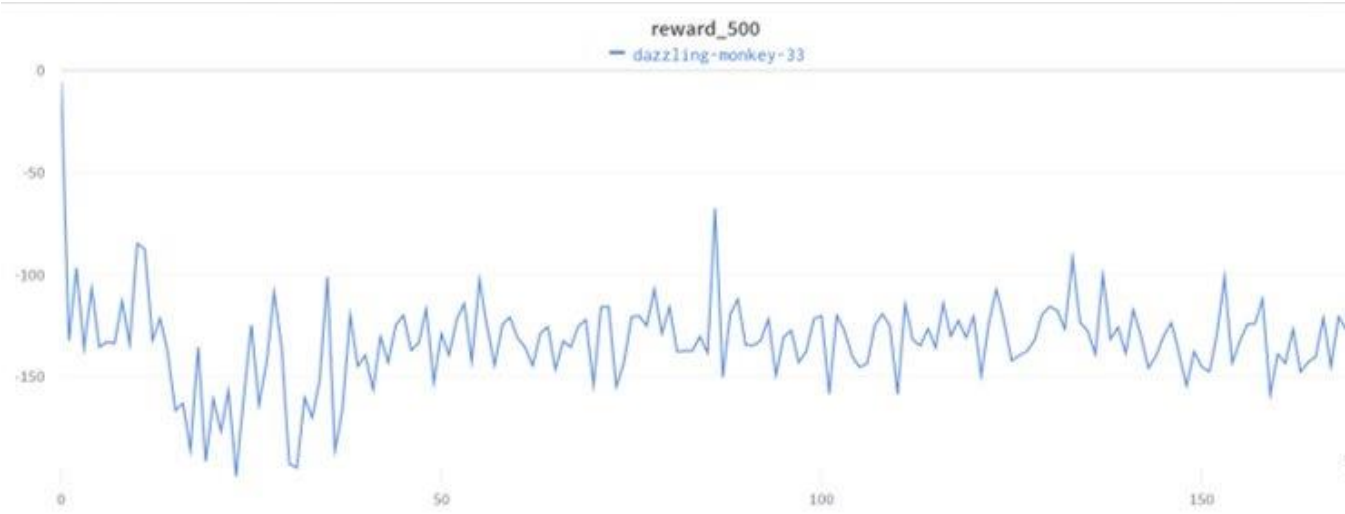

Figure 10 : Reward moyenne sur 500 steps (1ère fonction de perte, batch_size=50, gamma=0.9999)

2) On obtient un deuxième résultat avec la deuxième fonction de perte ou avec la première fonction de perte lorsque que le gamma est plus faible (à 0.99) ou que l'on augmente la taille de batch :

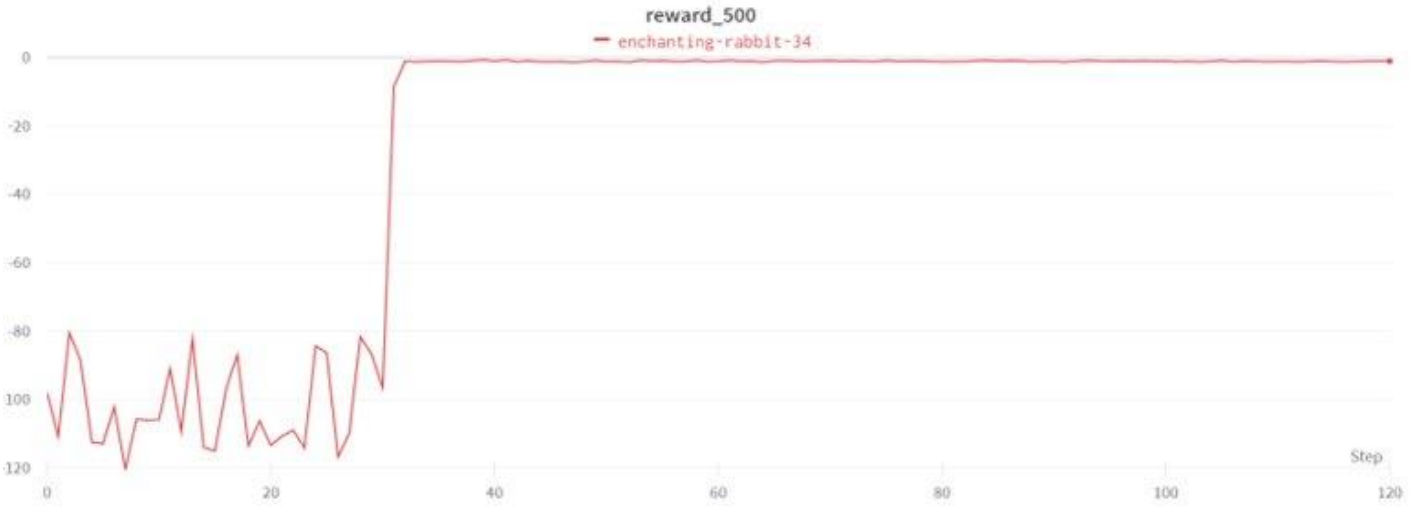

Figure 11 : Reward moyenne sur 500 steps (1ère fonction de perte, batch_size=100, gamma=0.9999)

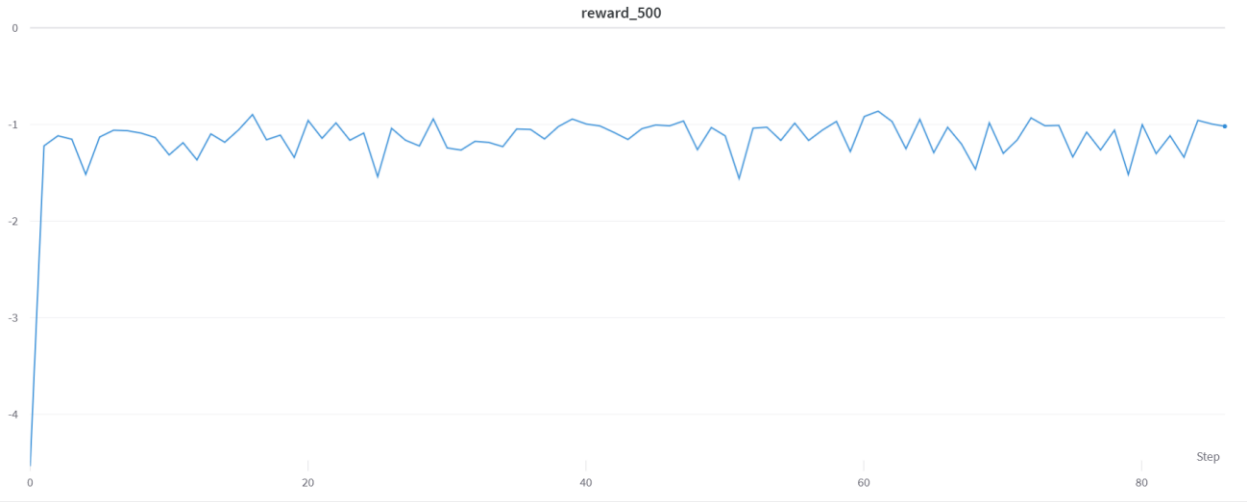

Figure 11 : Reward moyenne sur 500 steps (1ère fonction de perte, batch_size=50, gamma = 0.99)

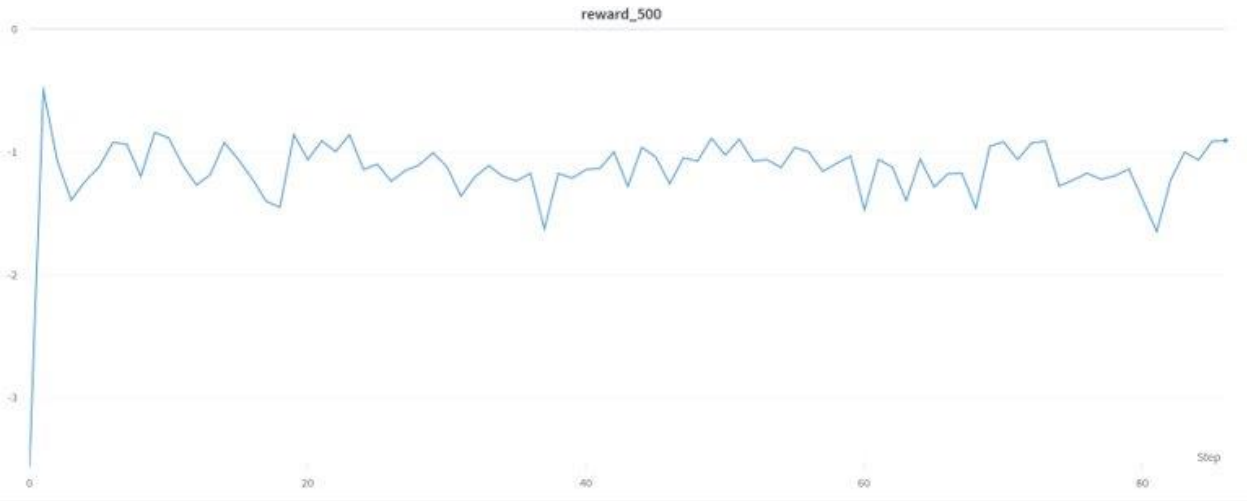

Figure 12 10: Reward moyenne sur 500 steps (2ère fonction de perte, batch_size=50, gamma=0.9999)

Ici, la reward tend vers une valeur un peu en dessous de zéro et l'environnement tend vers un état où toutes les éoliennes sont hors-services avec un agent qui prend seulement la décision de ne faire aucune correction. La reward est en-dessous de zéro car elle a été centrée, sinon elle serait logiquement à zéro puisque les éoliennes ne produisent plus et aucun frais de maintenance n'est engendré.

Ces résultats sont moins bons que ceux de l'agent de Q-learning car la reward reçue est moins élevée et l'agent apprend finalement à ne rien faire.

Possibles explications de ces résultats :

- Les entrées et sorties du réseau ne sont pas adaptées,

- Les fonctions de perte ne sont pas adaptées,
- L'exploration n'est pas bonne avec la politique epsilon-greedy et un epsilon décroissant,
- La structure du réseau de neurones est mauvaise (nombre de couches et de neurones),
- Les fonctions d'activations ne sont pas adaptées.

IV- Conclusion et perspectives

1) Valeur ajoutée

En termes de valeur ajoutée pour le client, notre travail abouti sur un code permettant de répondre à ses besoins, avec un environnement de taille restreinte. En effet, notre agent non-deep est fonctionnel sur un environnement allant jusqu'à 20 items, et même si nous n'avons pas pu évaluer ses performances sur un environnement aussi grand, nous avons pu constater qu'il est proche de la solution « optimale » (assimilée à celle du solver) pour un environnement de 3 items.

Il faudrait à présent essayer d'élargir le problème au deep pour pouvoir agrandir l'environnement étudié, et encore une fois, la valeur ajoutée de notre projet n'est pas nul à ce niveau-là. En effet, même si nous n'avons pas réussi à obtenir des résultats convaincants, nous avons pu fixer des pistes d'explications et d'améliorations pour de potentielles futures tentatives.

2) Fonctionnement du groupe

Dans l'ensemble nous sommes assez satisfaits du fonctionnement du groupe : nous avons réussi à tous prendre part au projet, à en comprendre le fonctionnement global malgré le fait que l'on se soit clairement séparé les tâches.

Malgré tout, il y a certaines choses à améliorer. En effet, certains se sont retrouvés à travailler individuellement sur certaines tâches, si bien que les autres n'ont pas vraiment pu en comprendre la finalité en profondeur, ce qui fait qu'ils étaient les seuls à vraiment pouvoir corriger leur code, et à répondre aux questions techniques. De plus, les disparités de connaissance en termes de ML et de niveau en termes de code n'ont pas arrangé les choses, obligeant certains à se former assez longuement avant de pouvoir espérer apporter quelque chose d'un point de vue ML.

3) Perspectives

Comme mentionné plus tôt, la principale perspective de ce projet serait d'élargir le problème à un environnement plus grand grâce au deep. Nous pourrons ainsi mettre en place un modèle nettement plus réaliste, avec plus d'actions et de fonctionnalités (comme le bridage que nous avons mentionné).

Parmi ces dernières, nous avons réfléchi à de nombreux moyens de rendre notre modèle plus réaliste, et plus adaptable à une vraie situation. Par exemple, il pourrait être intéressant de séparer les états de panne (comme mentionné dans la présentation de l'environnement) selon les différents équipements de l'item, et ainsi adapter les corrections en fonction de l'item en panne. Une autre perspective serait d'adapter la politique aux saisons : la réussite d'une mission de correction (préventive comme corrective) peut dépendre de la météo qui dépend elle-même de la saison. Ainsi, si l'agent juge que la météo des prochains jours risque de menacer la réussite d'une correction, il attendra avant de l'effectuer.