


UFR 6
Université Paul Valéry, Montpellier III

Rapport d'alternance

Rapport d'alternance de mi-parcours

Alvin Vedel

Janvier 2024

Remerciements

Je tiens à remercier toutes les personnes que j'ai rencontré et dont j'ai énormément appris lors de cette année de Master 1 MIASHS en alternance, que ce soit en entreprise au sein du Groupe Capelle ou lors de mon stage à la maison de la télé-détection.

Mes remerciements vont tout particulièrement à Jérôme Pasquet qui m'accueille en tant que stagiaire et me forme de manière complète et approfondie. Mais aussi Camille Marguerit qui m'a encadré lors de mon alternance en entreprise.

Je remercie également les enseignants du Master MIASHS à l'Université Paul Valéry, en particulier Mme. Sophie Lèbre et M. Maximilien Servajean, les responsables du Master, pour leur dévouement envers les étudiants. Je tiens aussi à exprimer ma gratitude envers M. Laurent Piccinini, mon tuteur universitaire, qui veille attentivement au bon déroulement de mon apprentissage. L'équipe enseignante s'est toujours tenue à disposition pour répondre à mes interrogations et m'ont donné l'envie d'en apprendre toujours plus sur le merveilleux monde de la data science.

Cette année marque une étape majeure dans mon orientation et sur les choix qui s'offrent à moi à la fin du Master, c'est pourquoi je tiens à remercier encore une fois M. Pasquet qui nourrit mes ambitions et m'encourage à fournir des efforts à la hauteur de celles-ci.

Merci sincèrement.

Table des abréviations

Acronyme	Nom Complet
AAG	Assistant Administratif de Gestion
ACM	Analyse des Correspondances Multiples
AFC	Analyse Factorielle des Correspondances
API	Application Programming Interface
BI	Business Intelligence
DAG	Directeur d'Agence
DB	Data Base
DWH	Data Warehouse
DOP	Directeur des Opérations
ETL	Extract Transform Load
HT	Hors Taxes
ISO	International Standardization Organisation
JSON	JavaScript Object Notation
MIASHS	Mathématiques et Informatiques Appliqués aux Sciences Humaines et Sociales
MS	Microsoft Server
PDI	Pentaho Data Integration
PIIC	Pole Industriel et Innovation Capelle
QSE	Qualité Sécurité Environnement
RSE	Responsabilité Sociétale des Entreprises
SMV	Séparateur Modulaires de Voies
SQL	Structured Query Language
TM	Traffic Manager
TTC	Toutes Taxes Comprises
TVA	Taxe sur la Valeur Ajoutée
UP	Unité de Production

TABLE 1 – Table des abréviations

Table des matières

Table des figures

Liste des tableaux

Introduction

La science des données est un domaine vaste qui est abordé dans le Master de Mathématiques et Informatiques Appliqués aux Sciences Humaines et Sociale (MIASHS) sous l'aspect des statistiques et du Machine Learning (ML). Ces deux grandes composantes sont connexes et se rejoignent sur bien des aspects avec au centre la donnée. Les informations sur lesquelles nous travaillons peuvent prendre différentes formes, dans le cadre d'une entreprise l'objet d'intérêt est souvent relatif à des clients qui sont considérés comme des individus et sont identifiés par des variables. Afin de les comprendre il est possible d'extraire des statistiques descriptives ou les visualiser à l'aide d'outils comme Tableau ou Power BI. Il est également intéressant d'extraire davantage d'informations que ce qui s'offre à nous et construire des modèles prédictifs permettant de comprendre le comportement de nouveaux individus, chose qui peut-être faite à l'aide de modèle statistiques ou de modèles de Machine Learning. Cela nécessite fréquemment des traitements au préalable sur ces données pour en tirer le meilleur parti.

Dans le cadre de mon stage à la Maison de la Télé-Détection (MTD), c'est principalement à l'aide du Deep Learning (DL) que nous analysons les données. Plus précisément, le Deep Learning est un outil qui nous permet d'effectuer des prédictions de haute précision pour qu'un agent accomplisse une mission. Le réseau n'est qu'une modélisation de la politique de décision qui mène au succès, il s'agit d'un paradigme du Deep Learning : le Deep Reinforcement Learning (RL). Les données sont cette fois une observation de l'environnement dans lequel évolue l'agent et bien que cela soit très différents de mes tâches effectuées en entreprise, une forme d'universalité de la donnée persiste à travers les traitements mis en place et les technologies employées.

Ce mémoire d'alternance est l'occasion de retracer les projets auxquels j'ai contribué au long de l'année passée en présentant rapidement mon parcours au sein de l'entreprise Capelle et en mettant l'accès sur mon stage de recherche à la Maison de la télé-détection.

Chapitre 1

Alternance au sein du groupe Capelle

Sommaire

1.1 Présentation de l'entreprise

1.1.1 La fondation du groupe

La société de transport Capelle basée à Vézénobres a été fondée en 1951 par René Capelle. Unique conducteur de l'entreprise, il proposait des prestations de transport conventionnel en acheminant du charbon en provenance des Houillères du bassin Cévenol. C'est dans les années 1970 que la société se diversifie dans le transport exceptionnel, notamment avec le programme nucléaire. Lors des années 1990, le réseau de l'entreprise se développe à travers différentes agences partout en France et elle reçoit le palmarès du Transporteur de l'année, une véritable reconnaissance professionnelle. La société commence à s'ouvrir aux marchés européens par exemple en Espagne et au Portugal. En 1999, elle rachète la moitié de la société TSAG et atteint les 200 salariés. Les années 2000 marquent un tournant pour la société avec la signature de gros contrats comme avec AIRBUS, le rachat de nombreuses sociétés de transports, l'agrandissement des effectifs qui atteignent les 700 employés ainsi que la formation du Groupe Capelle. L'hégémonie du Groupe Capelle se concrétise dans les années 2010 en devenant le leader français du transport exceptionnel. Le groupe s'élargit à travers la création de nouvelles agences en France mais également en Europe avec des agences en Espagne, au Luxembourg, au Royaume-Uni, en Pologne, au Pays-Bas et en Allemagne jusqu'à s'imposer en tant que premier transporteur exceptionnel d'Europe. Le groupe compte actuellement plus de 2000 collaborateurs pour 45 sites répartis en Europe 1.1, une flotte de plus de 1000 tracteurs et 2000 semi-remorques fournissant leurs services à plus de 10 000 clients.

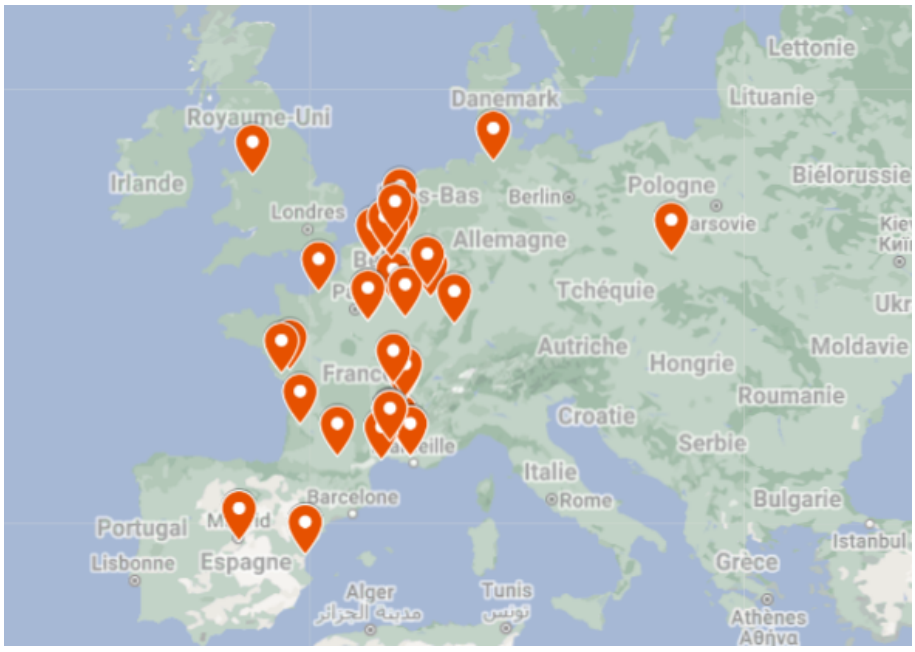


FIGURE 1.1 – Répartition des agences Capelle.

1.1.2 Le fonctionnement réel

Le fonctionnement de l'entreprise est le suivant : un client passe une commande au près d'un commercial, cela implique la création de devis dans le logiciel ComOn développé en interne. Il est ensuite transformé en commande à laquelle on associe un tiers, le client, et une facture. Ces informations sont stockées dans des bases de données Oracle hébergées dans le datacenter de la filiale ESI. Un service gère les autorisations nécessaires au transport conventionnel et le service du Dispatch l'assigne à une agence. Les commandes sont alors dupliquées dans le logiciel externe Open de la société CJM et stockées dans des bases Microsoft SQL avec les informations du voyage associé. Ces bases contiennent également les informations sur le planning, la facturation et les véhicules de l'entreprise. Au niveau des agences, un Traffic Manager (TM) gère administrativement le transport en communiquant avec le conducteur responsable du voyage. De nombreuses vérifications sont nécessaires et effectuées par les Assistants Administratifs de Gestion (AAG). Un suivi des activités conducteurs est possible via l'application TruckConnect développée en interne par ESI ce qui nous permet de produire des statistiques. Les données récupérées ainsi sont stockées dans des bases Oracles différentes de ComOn, hébergées sur les serveurs ETL ou DWH. Une fois le transport effectué, le service facturation émet la facture et en cas de contentieux le service de Credit Management prend la relève. Une fois la facture payée, ses informations atterissent dans le logiciel externe CEGID dans des bases Microsoft SQL Server. Ce logiciel comporte également des informations liées aux ressources humaines car il gère les payes et fonctionne de pair avec SOLID, également un logiciel externe relatif aux cartes de conducteurs. Les Directeurs d'Agence (DAG) supervisent

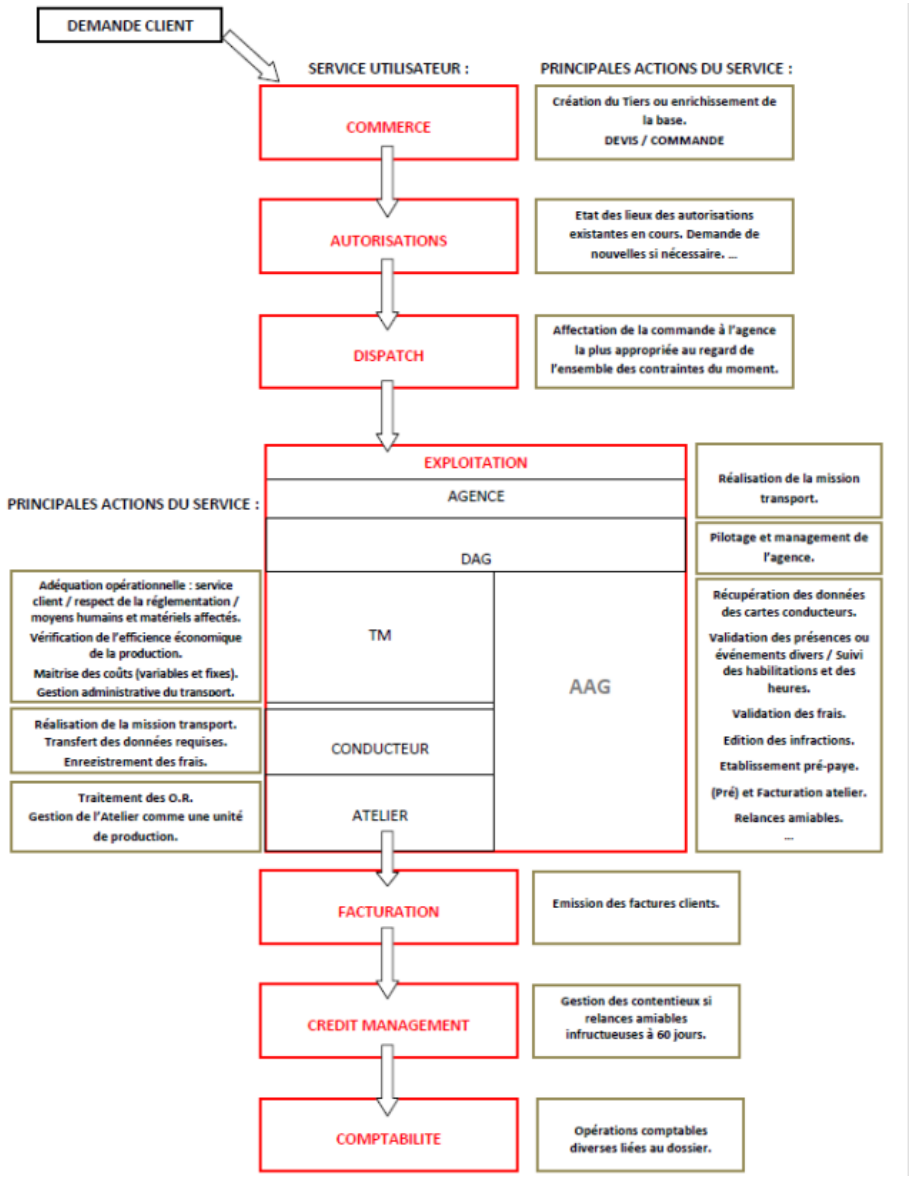


FIGURE 1.2 – Schéma du suivi d'un dossier.

cela et rendent des comptes aux Directeurs des Opérations (DOP) qui gèrent des régions et fixent des objectifs pour leurs agences.

1.1.3 Engagements RSE

Le groupe Capelle s'engage dans la Responsabilité Sociétale des Entreprises (RSE), notamment via la charte Qualité Sécurité Environnement (QSE) qui couvre ces trois domaines. Concernant l'environnement, les objectifs incluent la maîtrise de la consommation de gazole et des émissions de CO2, la réduction des risques de pollution chimique et la gestion des déchets. Cette charte, signée en 2021 par le président du directoire, les DOP, le directeur du site d'Alès et le directeur du service QSE, formalise ces engagements. Le groupe a obtenu la norme ISO 14001 pour son management environnemental en 2015, ainsi que la norme ISO 9001 pour son organisation et fonctionnement optimisés la même année. La filiale ESI, spécialisée dans l'hébergement de données, possède les certifications ISO 27001 pour la sécurité des données et Véritas pour les données de santé. Ses serveurs de Tier III garantissent une haute disponibilité sans interruption pour maintenance. Bien que le site de la filiale ne mentionne pas d'autres certifications environnementales, elle bénéficie aussi de la norme ISO 14001. Des mesures comme le refroidissement hydraulique pourraient être envisagées, conformément aux enjeux environnementaux et sociétaux du numérique discutés en cours. Le service QSE surveille le respect des engagements à tous les niveaux, avec des outils de suivi des taux de CO2 et de consommation de gazole. Les rapports mensuels indiquent une amélioration continue dans les domaines de la qualité, de la sécurité et de l'environnement.

1.2 Mon rôle dans l'entreprise

1.2.1 Définition du terme BI

Le service de Business intelligence ou BI a pour but de centraliser les données de la société, les rendre accessibles aux personnes concernées et en tirer le maximum d'information de la manière la plus efficace possible. Un tel service répond également à la problématique de la masse de données que possède l'entreprise, l'intégralité ne pouvant être traitée avec des solutions basiques comme Excel. Le développeur BI traite donc toutes sortes de données relatives à l'entreprise comme introduites dans la section précédente, on peut les séparer en 6 grands thèmes : le commerce, la finance, l'exploitation, le parc, les ressources humaines et la QSE. Il s'agit donc de produire de la connaissance sur toutes les facettes de l'entreprise. Le service BI comportait 2 autres membres, notre cheffe et ma responsable d'apprentissage Camille Marguerit qui a fondé le service il y a plusieurs années de cela ainsi qu'un alternant de l'école des mines d'Alès Enzo Dardaillon.

1.2.2 Les données relatives au monde du transport

Comme mentionné précédemment, les données exploitables sont multiples. On retrouvera des données de commerces, de devis, de commandes, de chiffres d'affaires, de facturation, de ressources humaines comme dans toute entreprise mais également des données plus spécifiques au monde du transport avec toutes les informations concernant les véhicules. Avec par exemple leur localisation : état du parc des véhicules, l'ensemble des contrôles qui leur sont liés, les données de conduites remontées par les boitiers TruckConnect. Les activités du groupe étant aussi liés à l'industrie, il existe également des données relatives aux ateliers de production et aux stocks en agences. Il ne s'agit là que d'une liste récapitulative mais le nombre de tables dans les bases de données est de l'ordre de plusieurs centaines.

1.2.3 Les missions du développeur BI

Le développeur BI intervient à toutes les étapes de la chaîne de traitement de la donnée. Il a pour mission de les récupérer de sources diverses : appels à des Application Programming Interface (API), extraction et jointures de bases de données, etc. Pour ce faire on utilise des Système de Gestion de Base de Données (SGBD), comme DBeaver, et Postman pour paramétrer les requêtes puis on définit une tâche de récupération automatisée à l'aide de Pentaho, l'outil d'Extract Transform Load (ETL). Cet outil permet également de définir la chaîne de traitement des données par un système de glisser-déposer avec plusieurs opérations à disposition. L'ensemble de ces tâches automatiques sont exécutés sur un ordinateur distant mais sont lancés à partir de Jenkins, un planificateur de tâche. Afin de versionner les traitements, un gitlab a été mis en place dans lequel les fi-

chiers "**.kjb**" ou "**.ktr**" de Pentaho sont archivés. Afin d'aller plus loin lors de mon passage dans l'entreprise, j'ai proposé des traitements faisant appel à des scripts Python ou R pour appliquer les enseignements de notre formation. Une fois les traitements qui récupèrent les données, les traitent et les enregistrent dans une nouvelle base, la mission du développeur BI est de les afficher à l'aide d'un outil de Reporting. Tableau est la solution possédée par l'entreprise, il permet l'affichage de graphiques simples tout en laissant une marge de créativité pour embelir l'affichage. Il permet également d'afficher des données brutes sous forme de tableaux croisés, une modalité souvent privilégiée par les commanditaires. En effet, le développeur BI répond à un besoin exprimé par un membre de l'organisme, généralement un décideur ou une personne qui a besoin de suivre l'activité de l'entreprise. C'est pourquoi il doit constamment échanger par mail ou lors de réunions pour préciser les attendus et s'assurer d'aller dans la bonne direction. Les demandes sont soumises via un système de "Tickets" développé par un membre du service et sont assignés à chacun par notre cheffe qui suit l'avancée de tous les projets à l'aide d'un agenda type Tom's Planner.

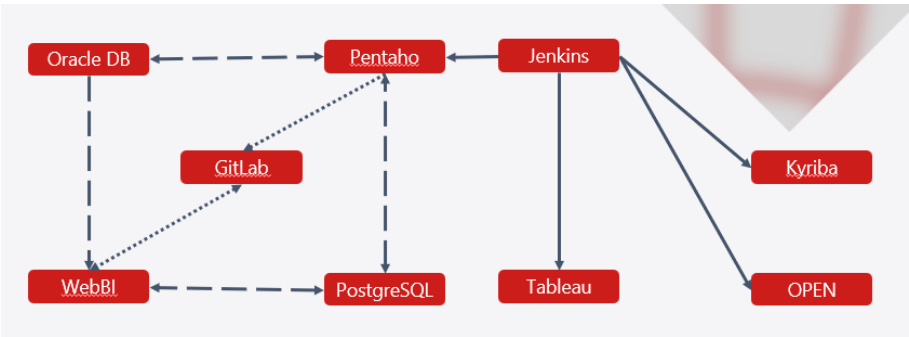


FIGURE 1.3 – Schéma des interactions entre serveurs BI tiré de la documentation des serveurs BI par Enzo Dardaillon. [1]

1.2.4 Un exemple typique de mission

Pour écourter la présentation sur l'alternance au profit du stage que j'aborderai dans le chapitre suivant, je me contenterai de décrire une mission représentative du travail accompli en tant que développeur BI.

L'objectif

Il s'agit d'une requête émanant de la filiale MecaIndustrie dont l'activité principale est le transfert d'industrie et la manutention. Leur directeur d'agence a exprimé le besoin de suivre les devis pour ses trois agences, cela comprend des informations comme le nombre de devis émis, de devis transformés, de devis en cours ou le taux de transformation. L'outil de suivi doit permettre une vue mensuelle des informations, à la fois par agence et par chargé d'affaire. Dans un second temps, le suivi des charges associées à ces devis doit être

possible ainsi que la facturation en cours et réalisée. Une version sous forme de fichier Excel existait déjà mais le développement en BI de l'outil a pour but de l'automatiser, assurer sa mise à jour quotidiennement et de fournir une visualisation plus esthétique et agréable à prendre en main. A cela j'ai ajouté de ma propre initiative un objectif prédictif afin d'estimer la probabilité de succès, d'abandon ou de perte des devis. C'est une information supplémentaire pouvant s'avérer pertinente lors du suivi de ceux-ci.

Les données

Les données relatives aux devis de la filiale sont hébergés sur un serveur SQL nommé X3, pour les récupérer j'ai repris un ancien traitement qui servait à alimenter le fichier Excel. En termes de variables, on possède pour chaque devis un nom de client et son identifiant, un champ "représentant" désignant le chargé d'affaire, la date de début, la date de fin, les montants Hors Taxes (HT) et Toutes Taxes Comprises (TTC), une liste d'articles ainsi que leurs catégories, le montant et la quantité de chaque article, l'agence d'exploitation et l'état du devis. L'extraction en question comporte une difficulté de traitement car des lignes se répètent pour permettre d'afficher les détails sur les articles. De nombreux champs doivent aussi être calculés pour correspondre au fichier Excel, j'ai simplement repris celles d'un fichier exemple.

Méthodes et outils

Avec l'outil d'ETL Pentaho [3] je procède à l'extraction des données depuis les bases X3 pour les dupliquer sur le serveur ETL. Un deuxième script Pentaho les écrit dans un fichier Excel qui sera ouvert par un troisème script, en Python cette fois pour traiter les données. A l'aide des librairies numpy [4] et pandas [5] je calcule les nouveaux champs cités précédemment ainsi que des champs pour permettre la lecture des données par Tableau [6], par exemple la répétition du champ montant provoquera une somme énorme en comparaison avec la vraie valeur lors de l'agrégation de champs. J'ai également modifié la structure du fichier Excel qui comportait une colonne par mois en ajoutant simplement une colonne "date valeur" avec comme modalités l'ensemble de ma plage temporelle. Réduisant drastiquement le nombre de colonnes, pénibles à glisser en Tableau, au prix d'une augmentation du nombre de lignes.La solution pour gérer les doublons a été de définir la première ligne de chaque devis avec les valeurs réelles et de mettre les variables quantitatives des autres lignes à 0, ce qui permet de sommer à la valeur attendue.

Une fois les opérations terminées, le script Python écrit les données sous formes d'un fichier csv. Il sera lu par un quatrième script Pentaho et inséré dans une base de données. Ne reste alors que la partie visualisation pour laquelle les attentes du commanditaires étaient très claires et ne laissaient pas beaucoup de place à l'imagination. Le Report s'organise en 6 onglets, deux pour les devis, deux pour les charges et deux pour la facturation avec des tableaux croisés et quelques diagrammes en barres qui facilitent la comparaison

entre les agences ou les chargés d'affaire.

En ce qui concerne l'objectif supplémentaire de prédiction, j'ai intégré le traitement dans le fichier Python en faisant appel à la librairie scikit-learn et en utilisant un algorithme de RandomForest comme vu en cours de classification. Nous sommes face à un problème de classification multi-classe dans lequel on essaie de savoir l'état du devis en fonction des autre variables. Plus précisément nous souhaitons prédire le statut final des devis en cours et déterminer s'ils seront "gagné", "perdu" ou "abandon". Pour cela un traitement supplémentaire des données s'impose avec un *one-hot encoding* des variables qualitatives, une transformation de certaines données quantitatives tel que la conversion des dates en valeurs numériques. Pour évaluer le modèle créé, je sélectionne un sous échantillon parmi les devis terminés qui fera office de jeu de validation et j'affiche les résultats dans une matrice de confusion. Le calcul de métrique comme l'accuracy me permettent également de valider le modèle, ici j'avais obtenu 90% sur le jeu de validation après avoir cherché le nombre d'arbres optimal. Après validation du modèle, j'en crée un nouveau à partir de l'ensemble des données pour qu'il possède plus d'informations en entrée et j'effectue mes prédictions sur les devis en cours en sortant la probabilité d'appartenir à chaque classe plutôt que la classe directement.

Les résultats

C'est donc avec des forêts de 100 arbres que l'on obtient les meilleurs résultats et selon le tirage aléatoire fait par la séparation entre jeu d'entrainement et jeu de validation l'accuracy fluctuent entre 85 et 90%. Lors de l'affichage de la matrice de confusion, on constate un déséquilibre des classes et cette accuracy est obtenu en prédisant beaucoup de "gagné" car il y en a beaucoup. Si on regarde les autres classes individuellement, on dépasse légèrement les 50% de vrais positif ce qui n'est pas loin du hasard. C'est pourquoi afficher les probabilités m'a paru être une solution plus honnête et qui convient mieux au besoin de précision du commanditaire.

Chapitre 2

Le Stage de Recherche

Sommaire

2.1 Introduction

Après 5 mois de travail au sein de l'entreprise Capelle, je ressentais le besoin de trouver une alternance davantage en adéquation avec la formation proposée dans le Master et j'ai accepté un stage à la maison de la télé-détection sous la supervision de M. Pasquet à partir du 18 mars.

Il s'agit d'un stage de recherche sur le Deep Reinforcement Learning qui a pour objectif le pilotage d'une flotte de robot sous-marin. Le sujet est proposé par la Direction Générale des Armées (DGA) et les robots doivent trouver des mines, ils permettront ainsi de les désamorcer et d'éviter aux navires français d'être endommagés par l'explosion. Cela pouvant être une thématique de recherche trop ciblée, notre objet d'intérêt se porte sur une course de robots devant faire des tours de piste. Pour situer la problématique de recherche il convient de définir le principe du Deep Reinforcement Learning (Deep RL). Le RL "classique" consiste à laisser un agent interagir avec un environnement, il reçoit une information de celui-ci et choisit une action optimale lui permettant d'atteindre un objectif. La particularité du Deep RL est que le choix de l'action se fait à l'aide d'un réseau de neurones ce qui permet d'avoir une politique d'action avec beaucoup de paramètres et une meilleure généralisation. Lors de l'entrainement du modèle, on définit une fonction de récompense (*reward*) qui donne une indication à l'agent sur l'atteinte de son objectif, il va chercher à maximiser cette récompense. Une fois l'action effectuée, l'agent reçoit un reward et une nouvelle observation de l'environnement qui a été impacté son choix. Ces itérations successives d'entrainement sont des pas (*step*) qui s'inscrivent dans un épisode. On considère ces épisodes comme des parties d'un jeu et la convergence des poids du réseau se fait sur un nombre de parties relativement élevé.

Mais le principe du Deep RL est loin d'être nouveau, ce qui motive ce sujet de recherche est davantage lié aux spécificités de notre problème. Premièrement il s'agit d'un problème de régression car nous opérons dans un espace continu et que l'on souhaite prédire des valeurs dans $\mathbb{R}$, à la fois pour l'angle de l'orientation du robot et sa vitesse. Discrétiser le problème pourrait être une solution mais reviendrait à abandonner une solution optimale au profit d'une solution simple, ce n'est pas ce qui nous intéresse. Le deuxième aspect qui nous préoccupe est la co-existence d'agents indépendants qui doivent suivre une formation au gré des utilisateurs finaux. On peut déjà supposer qu'introduire plusieurs agents indépendants dans un espace fortement non-markovien pourrait causer des problèmes. Les forcer à respecter une formation tout en considérant un objectif et des obstacles en est un autre.

Pour résoudre ce problème complexe, une revue de la littérature s'impose et nous permettra de mettre en place des expériences dans un environnement simulé. Nous détaillerons notre avancée dans le projet et les perspectives pour la suite du stage.

2.2 Revue de la littérature

- 2 algos dans DRL : principes et avantages/inconvénients => Comprendre les bases - Problématique du continu : DDPG - Optimiser le Replay Memory : Prioritzed - Gérer plusieurs tâches en même temps : multitask - Optimiser les temps de calcul : parallélisation

2.2.1 Le Q-Learning

Le Deep Reinforcement Learning, ou apprentissage par renforcement profond, est divisé en deux familles principales. Le Deep Q-Learning proposé par Mnih et al. en 2013 [7] considère une fonction $Q^*(s,a) = \max_\pi \mathbb{E}[R_t \mid s_t = s, a_t = a, \pi]$, avec s l'état, a une action et π la politique de décision de l'agent. R_t est une variable aléatoire qui représente l'espérance cumulée décomptée d'un facteur γ de la récompense au fil du temps $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$. Le réseau qui représente la politique π et dont on note les poids θ peut être mis à jour en minimisant une séquence de fonctions de pertes $L_i(\theta_i)$ définit comme $L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$. Et $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a\right]$ étant la cible à l'itération i. Pour récapituler on choisit l'action qui maximise une fonction Q, ce qui revient à maximiser l'espérance finale des gains pour l'agent. Mnih et al. introduisent également le concept de mémoire et stockent un nombre finies de transitions qui pourront être rejouées pour la mise à jour des poids. Une transition se note sous la forme d'un tuple (s,a,r,s') avec s' l'état suivant l'action et r la récompense associée à l'action.

La gestion de cette mémoire est une composante importante qui a été sujette à de nombreuses recherches, on retrouve par exemples des architectures utilisant des Long Short Term Memory (LSTM) comme dans les travaux de Hausknecht et al. en 2015 [8]. Ils ont proposé une variante du Deep Q-Network (DQN) de Mnih et al. nommée le Deep Recurrent Q-Network (DRQN). Leur objectif était d'introduire une instabilité dans l'environnement d'entrainement de l'agent, dans un simulateur sans aléa l'environnement est dit *Markovien* car on sait précisément quel sera l'état suivant après une action. Les environnements Markoviens ne correspondent pas à la réalité terrain et les environnements dans lesquels nous opérons sont dits *non-Markoviens*. Pour introduire l'instabilité, ils ont définit une probabilité de masquage de l'environnement passant d'un Markov Decision Process (MDP) à un Partially Observed Markov Decision Process (POMDP) Le DRQN a montré de meilleurs résultats que les DQN sur lors de leurs expériences sur les jeux Atari, cependant cette piste a été écartée par M. Pasquet pour leur manque d'efficacité.

Un autre point intéressant du Q-Learning qui a été montré par Wang et al. en 2015 [9] est qu'il est possible d'estimer deux fonctions distinctes qui servent à prédire la valeur de la fonction Q. D'une part la fonction Avantage $A^\pi(s,a)$ et de l'autre la fonction Valeur $V^\pi(s)$ qui s'additionnent pour obtenir $Q^\pi(s,a)$, cela entraine une modification de l'architecture du DQN que l'on peut observer dans la figure 2.1. Le réseau est séparé en deux flux qui suivent des paramétrisations distinctes notées α et β, le reste des paramètres est toujours noté γ. L'idée vient du fait qu'une action n'est pas toujours nécessaire et qu'elles n'ont

pas de répercussions sur les états suivants, le premier flux va estimer la valeur $V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)]$ et la seconde l'ensemble des Avantages $A^\pi(s,a)$ pour chaque action tel que $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. On peut estimer la valeur de l'avantage comme $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. C'est une différence d'architecture qui ne modifie pas l'algorithme d'apprentissage du DQN mais qui y est complémentaire permettant une amélioration des performances.

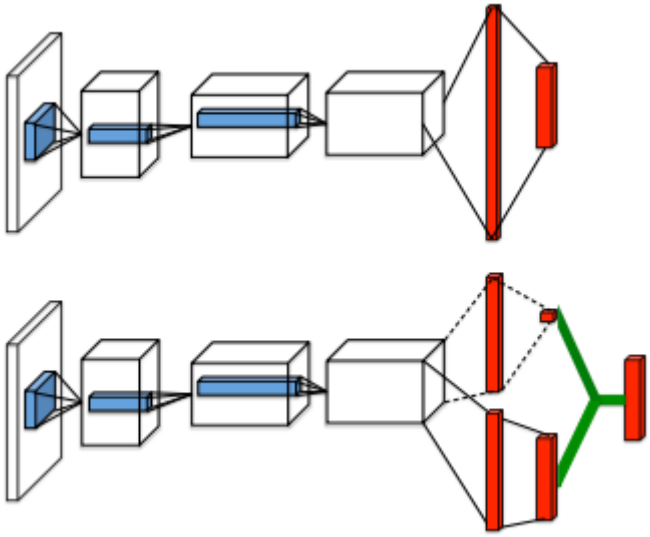


FIGURE 2.1 – Le DQN (en haut) et le DDQN (en bas)

La gestion de la mémoire peut être un facteur important pour la mise à jour des poids, en 2016 Schaul et al. [10] décrivent deux méthodes de priorisations des expériences applicables au DQN. L'objectif est de mettre en avant les expériences les plus "marquantes" ou "surprenantes" pour que l'agent en tienne compte. C'est particulièrement pratique dans notre cas ou les récompenses positives sont rares, au cours d'un épisode l'objectif n'est atteint qu'au plus une fois alors il est important de lui indiquer que cette expérience est significativement importante. Notamment dans le cadre d'une mémoire de taille limitée, un évènement important pourrait être amené à disparaitre sans avoir été rejoué. Les deux méthodes qu'ils proposent fonctionne sont les suivantes : la version *proportionnal* attribue à chaque transition une Temporal Difference (TD) error notée $\delta_j = R_j + \gamma_j Q(S_j, \arg\max_a Q(S_j,a)) - Q(S_{j-1}, A_{j-1})$ et attribut un poids p_j à la transition correspond à la valeur absolue de la TD-error. Dans la deuxième version, ils trient les transitions par TD-error et calculent le poids p_i comme $\frac{1}{\text{rank}(i)}$. La probabilité de tirage des transitions n'est alors plus uniforme mais définit par $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ avec α le paramètre d'importance de la priorisation (0 équivalent à une répartition uniforme).

Plusieurs points importants abordés au travers des précédents articles sont bons à noter, premièrement l'utilisation de réseaux *main* et *target* qui introduisent une grande stabilité dans l'apprentissage. L'action est choisie par le réseau *main* et la mise à jour des poids se fait sur le réseau *target*, après un nombre donné et suffisament grand d'itérations les poids sont synchronisés. Deuxièmement, l'agent a besoin d'explorer l'environnement

pour découvrir de nouvelles actions c'est pourquoi il effectue des actions aléatoires au début ce qui lui permet de remplir sa mémoire pour mettre à jour les poids. En Q-Learning l'agent choisit toujours l'action optimale alors sans exploration il reste coincé dans une politique non-optimale. Pour trouver un compromis optimal entre exploration et exploitation on définit une politique dite *greedy* avec un paramètre ϵ qui définit la probabilité du choix aléatoire. Cette probabilité est décroissante au fil des itérations mais on veille à la maintenir supérieure à 0 pour que l'agent explore tout au long de l'apprentissage.

2.2.2 Le gradient de politique

La deuxième grande famille d'algorithmes en Deep RL sont les algorithmes à gradient de politique. La principale différence avec le Q-Learning est qu'au lieu d'estimer la qualité d'une action avec une sortie linéaire dans les réseaux, on estime la probabilité qu'une action soit la bonne avec une fonction *softmax* en sortie du réseau. L'avantage de cette méthode est qu'elle optimise directement les paramètres de la politique sans avoir à apprendre les valeurs d'états ou d'actions. Un estimateur commun de gradient de la politique peut être défini comme $\hat{g} = \hat{E}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) A_t\right]$.

Plusieurs travaux ont été menés pour fournir des algorithmes garantissant l'amélioration de la politique, l'article de Schulman et al. en 2015 [11] sur les Trust Region Policy Optimization (TRPO) en est un parfait exemple. Ils définissent la somme décomptée d'un facteur γ des récompenses pour une politique π comme $\eta(\pi) = \mathbb{E}_{s_0,a_0,\ldots}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t)\right]$ et expriment cette même somme pour une nouvelle politique $\tilde{\pi}$ en fonction de la fonction Avantage évoquée précédemment comme $\eta(\tilde{\pi}) = \eta(\pi) + \mathbb{E}_{s_0,a_0,\ldots\sim\tilde{\pi}}\left[\sum_{t=0}^{\infty} \gamma^t A^\pi(s_t, a_t)\right]$. Sans trop rentrer dans les détails ils arrivent à garantir une amélioration de la politique à chaque mise à jour. Un tel algorithme nécessite l'estimation à la fois des probabilités d'action et de la fonction Avantage qui se calcule comme la différence entre la fonction Q et la fonction V. C'est pourquoi ils utilisent un système Acteur-Critique comme dans l'algorithme du Proximal Policy Optimization (PPO) également de Schulman et al. en 2017 [12] [13].

Les premiers à aborder la notion d'Acteur critique sont Lillicrap et al. en 2015 [14] dans le cadre d'un problème continu. Ils combinent les principes du Q-Learning et du gradient de politique bien que le choix de l'action soit déterministe et non stochastique. Le réseau acteur responsable du choix de l'action est entrainé par gradient de politique et l'objectif du réseau critique est de fournir un retour d'information sur l'action en estimant sa Q-value. C'est donc un concept important car il intervient à la fois dans les problèmes continus déterministes et de gradient de politique (stochastiques).

Dans le même direction que le TRPO, certaines recherches de Schulman et al. en 2015 [15] ont pour objectif de fournir la meilleure estimation de la fonction Avantage. C'est le cas de leur publication sur le Generalized Advantage Estimator (GAE) dans laquelle ils fournissent des estimations pour le gradient de la politique $\hat{g}$ et $A(s, a)$ comme $\hat{A}^{\mathrm{GAE}}(\gamma, \lambda) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$ avec γ et λ des hyperparamètres permettant le compromis

entre biais et variance dans l'estimation. L'intérêt de cet article est également qu'ils solvent une tâche continue dans un espace de grande dimension.

2.2.3 Parallélisation de l'entrainement

L'entrainement de tels modèle pouvant s'avérer très longs, un pan de la littérature s'intéresse à paralléliser l'apprentissage sur les Central Processing Unit (CPU) et les Graphics Processing Units (GPU). Stooke et al. en 2018 [16] décrivent des méthodes permettant d'optimiser les ressources informatiques d'une machine pour réduire drastiquement les temps d'apprentissage. Leurs méthodes, bien que très basiques, ont l'avantage d'être applicables à tout type d'algorithme que ce soit le Q-Learning ou le gradient de politique. Chacun des CPU gère plusieurs instances de l'environnement qui s'exécutent en parallèle. Ces instances sont séparés en deux groupes, lorsque le premier groupe choisit une action et l'exécute, le second met à jour ses poids après le tirage d'échantillons de la mémoire. Ceci permet de maximiser l'utilisation du GPU qui alterne entre les deux groupes et chaque GPU interagit avec plusieurs CPU. L'optimisation du GPU peut être faite de deux manières différentes, ils distinguent une version synchrone dans le quel le gradient est calculé localement puis moyenné avec les autres GPU pour mettre ensuite les poids locaux à jour, d'une version asynchrone dans laquelle chaque GPU possède une version locale des poids et applique les mises à jour dans un modèle stocké dans la mémoire GPU. Une réserve est à garder sur la technique car même si les temps d'apprentissages se comptent en minutes, certains algorithmes basés sur le gradient de politique ne semblent pas réussir à converger lorsque le nombre d'unité parallélisé devient trop grand.

Des travaux du même acabit ont été menés par Horgan et al. en 2018 [17] avec pour objectif de faire communiquer les mémoires des apprenants comme le montre la figure 2.2. Le modèle intitulé Ape-X fait appel au concept d'expérience priorisé de Schaul et al. [10] pour calculer des priorités de replay des transitions localement dans des acteurs. Ces acteurs représentent des agents qui interagissent avec des environnement distincts, ils effectuent une transition, calculent le poid associé et la stocke dans la mémoire commune. L'apprenant en tire un batch et met à jours ses poids puis recalcule les probabilités de tirage des transitions. Périodiquement il met à jour les poids des réseaux des acteurs qui possèdent une copie du réseau apprenant. Dans leur configuration, un acteur occupe un CPU et l'apprenant un GPU mais la configuration peut rapidement devenir coûteuse si l'on souhaite augmenter le nombre d'agents parallélisés.

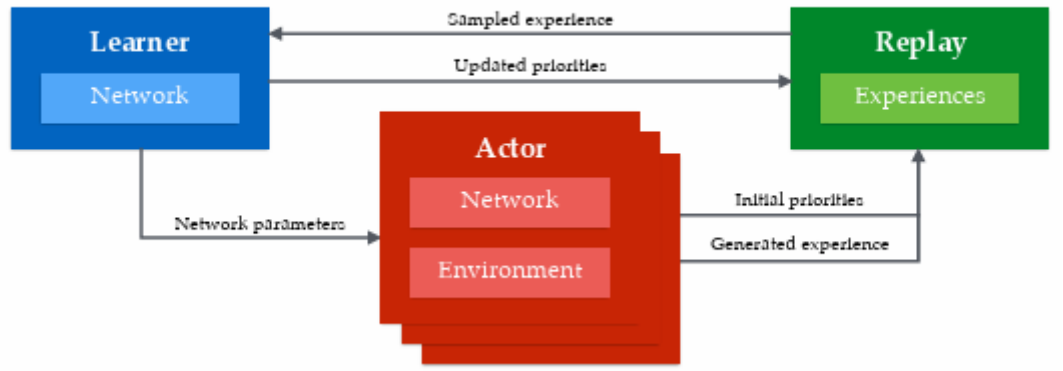


FIGURE 2.2 – Schéma de Ape-X

2.2.4 Agents multi-tâches

Les comportements attendus de nos agents n'étant pas limité à une simple tâche, il est important de se renseigner sur la façon dont les problématiques multi-tâches sont abordées en Renforcement Learning. En 2017, Cabi et al. présentent un agent qui résout une tâche "intentionnelle" tout en apprenant à résoudre des tâches complémentaires dites "involontaires". L'agent optimise plusieurs politiques simultanément mais le côté involontaire intervient dans le fait que les politiques complémentaires à la tâche principale sont apprises "off-policy", c'est à dire que le choix de l'action ne dépend pas de ces politiques là. Leur sujet d'étude concernait un agent opérant dans un espace continu à plusieurs dimensions et ils ont opté pour un algorithme de politique de gradient. Ils définissent une multitude de fonctions de récompenses de manière automatique qui correspondent à des interactions de l'agent avec les éléments de l'environnement ce qui peut être particulièrement intéressant pour gérer les interactions des robots de notre flotte entre eux. Le gradient de l'acteur est calculé comme l'espérance de la somme des gradients de politique : $\nabla_\theta J(\theta) \approx \mathbb{E}_{\rho^\beta}\left[\sum_i \nabla_\theta \mu_\theta^i(s) \nabla_{a^i} Q^\mu(s,a^i)\Big|_{a^i=\mu_\theta^i(s)}\right]$. Ils montrent également que plus l'agent optimise de récompenses simultanément plus son apprentissage est rapide et que dans certains cas un agent qui n'optimise qu'un seul objectif ne réussit pas du tout la tâche demandée.

L'inconvénient de cet algorithme est que la relation entre les récompenses involontaires et la principale n'est pas toujours explicite. A l'inverse, Jadenberg et al. en 2016 proposent l'agent UNREAL qui accomplit des tâches non-supervisée directement complémentaires de sa tâche principale. La tâche principale est apprise par gradient de politique avec l'algorithme Asynchronous Advantage Actor Critic (A3C) de Mnih et al. [18]. Les tâches auxiliaires, qui sont entrainées par Q-Learning, ont un rôle de contrôle d'une part avec une tâche de *Pixel Control* qui maximise la diversité de l'input visuel de l'agent et une tâche de *Feature control* qui maximise l'activation des neurones des couches cachées du réseau recevant l'information visuelle. Deux autres tâches auxiliaires sont présentées

permettant d'obtenir des *features* de meilleures qualité et une meilleur estimation de la fonction Valeur. Pour ce faire ils entrainent l'agent à prédire l'apparition de récompenses et partagent les features découvertes avec le réseau principal. Pour l'estimation de la fonction valeur, ils régressent une fois supplémentaire la valeur de $V(s)$ avec les features nouvellement apprises. Ces stratégies peuvent s'avérer très intéressantes pour notre problème d'exploration en trois dimensions dont les observations sont plutôt réduites et les récompenses rares.

2.3 Les expériences réalisées

Les expériences réalisées correspondent aux différents algorithmes implémentés dans le cadre du projet de Renforcement Learning. L'ensemble des codes est en Python et fait appel aux librairies Tensorflow et Keras pour les modèles de Deep Learning et à Numpy pour l'optimisation des calculs. Certaines librairies comme Matplotlib ou PIL ont été utilisé pour produire un retour graphique des épisodes de l'agent. L'ensemble du code est archivé sur un repository github ainsi que sur la machine distante mise à disposition par M. Pasquet sur laquelle tournent les scripts, cela permet de profiter de la puissance de calcul de 4 GPUs et 16 CPUs.

2.3.1 Débuts du RL

Il est important de préciser que l'on part de rien pour entrainer notre agent, cela implique que la création de l'environnement et de l'environnement nous incombe. Nous verons dans cette première partie la création des classes "Robot" et "Environnement" ainsi que la structure de l'algorithme d'entrainement.

Création de l'environnement

Pour modéliser l'environnement de notre problème, nous nous rapportons à un espace 2D dans lequel figure un objectif, un robot et un chemin pour y parvenir. Les autres cases sont considérées comme des obstacles pénalisant le robot. La carte correspondante à cette description visible sur le figure 2.3 est créée avec la librairie PIL, le robot est identifiable par le flèche noire et l'objectif par le disque blanc, le chemin en vert et les obstacles en rouge.

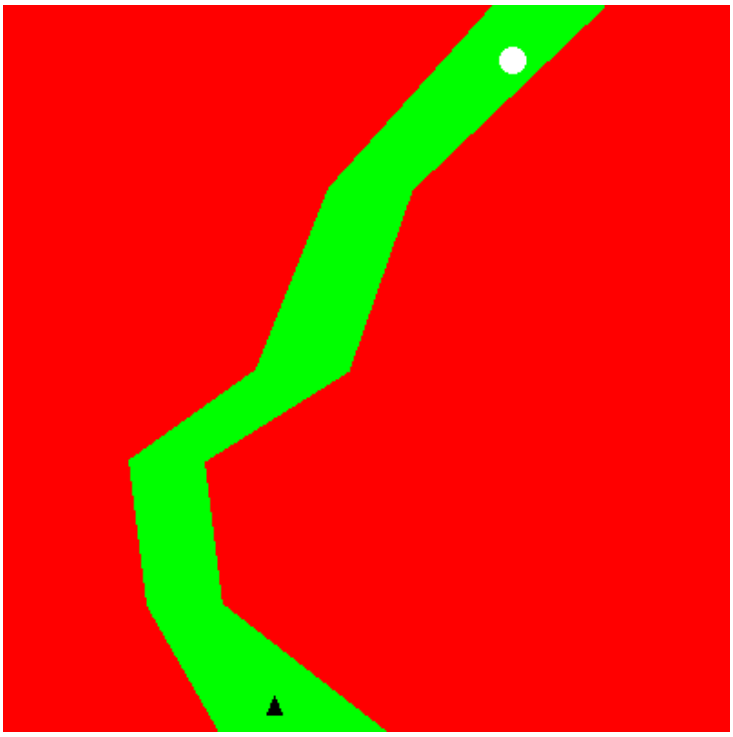


FIGURE 2.3 – Première carte

La carte est stockée en tant qu'attribue dans une classe *environnement* et possède une taille de 400 × 400. L'agent, qui est une classe à part que nous détaillerons plus tard, est également stocké dans l'environnement en tant qu'attribut, idem pour la position de l'objectif. Plusieurs fonctions sont essentielles au déroulement de l'algorithme :

- Fonction *reset* : restaure l'environnement dans son état initial (celui de la figure 2.3).
- Fonction *compute_reward* : calcule le *reward* (ou récompense) après une action de l'agent.
- Fonction *step* : prend en entrée une action de l'agent, calcule le nouvel état de l'environnement et assigne un *reward*, définit également un booléen *done* si l'agent a atteint l'objectif.

L'agent quant à lui possède comme attribut une position (coordonnées x, y), une vitesse, une orientation et un réseau de neurones. Il possède des attributs graphiques comme sa hauteur et sa largeur ainsi que des hyper-paramètres tels que $\epsilon, \epsilon_{\max}, \epsilon_{\min}$ le facteur de décroissance de la part de hasard dans ses actions, un hyperparamètres pour la taille de sa mémoire, la taille maximum de la mémoire, etc. Il possède également des matrices numpy permettant de stocker les transitions avec une matrice par type d'information : les états précédents et suivants, les actions choisies, les récompensent, les *done* et les informations du robots avant/après l'action qui vont de paires avec les états. Les fonctions qu'il possède permettent également le fonctionnement de l'algorithme :

- Fonction *rotate_robot* : permet de changer son orientation entre 4 valeurs (0, 90, 180 ou 270 degrés)

— Fonction *changer_ vitesse* : permet de changer sa vitesse entre 4 valeurs (0, 1, 2, 3) qui correspond au nombre de case pas avancement
— Fonction *avancer* : calcule la nouvelle position après changement des attributs et ajoute une "vague" pour introduire un aléa dans l'environnement qui décale l'agent
— Fonction *replay* : qui permet de piocher un batch aléatoire d'éléments dans la mémoire
— Fonction *remember* : stocke une transition dans la mémoire
— Fonction *choose_ action* : qui reçoit en entrée l'état de l'environnement et choisit une action, que ce soit de manière aléatoire ou par le réseau.

Plusieurs précisions s'imposent, le réseau de l'agent reçoit en entrée deux types d'informations. La carte sous la forme d'une image de 400×400 et les informations du robot sous la forme d'un vecteur de quatre éléments : sa coordonnée x, coordonnée y, sa vitesse et son orientation. La carte est encodée avec 4 valeurs différentes pour chacun des éléments, le réseau *one-hot encode* ces valeurs pour créer un cube de $400 \times 400 \times 4$ et ajoute deux dimensions pour stocker les valeurs de vitesse et d'orientation dans chaque pixel de la carte. Ces valeurs sont à 0 partout sauf aux coordonnées du robot. Une fois notre cube de $400 \times 400 \times 6$ obtenu, il passe par un réseau de convolutions et couches Dense dont les détails sont rapportés en annexe A.1.

L'algorithme

Parmi les deux familles d'algorithme, c'est le Q-Learning qui a été privilégié dans ce premier temps, celui implémenté s'inspire très largement du code fourni dans la documentation Keras [19]. Il se déroule dans deux boucles "for", la première qui permet d'effectuer les épisodes, soit l'équivalent d'une partie, et la deuxième pour les *steps*, c'est à dire la succession d'action de l'agent dans une partie. A chaque début d'épisode on réinitialise l'environnement et la boucle des steps commencent. Un nombre maximal de step, ici 600, est défini pour éviter que l'agent ne remplisse sa mémoire de mauvaises expériences s'il ne parvient pas à atteindre l'objectif. Le déroulement d'une itération de step est assez simple, l'agent observe l'environnement et choisit une action, il l'exécute et reçoit un reward et un état done. Si le booléen done est vrai alors on termine la seconde boucle et un nouvel épisode commence. Les transitions ne sont pas rejouées à chaque itérations pour limiter les calculs mais toutes les quatre actions. On désigne cette séquence sous le terme d'*Experience replay* [7], les nouvelles valeurs de la fonction Q sont estimées et les poids du réseaux sont mis à jour.

Une première différence à noter avec l'algorithme Keras est qu'avec notre modèle à deux sorties, on estime deux fonctions Q, une pour l'angle et une pour la vitesse. Elles sont donc traitées parallèlement et on obtient deux *loss* que l'on somme à la fin pour estimer l'erreur totale du réseau.

Concernant les hyper-paramètres, la fonction de perte utilisée est la loss Huber qui

convient aux problèmes de régression et définie comme :

$$\text{Huber}(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{si } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{sinon} \end{cases}$$

L'apprentissage s'effectue avec un optimiseur Adam et un pas d'apprentissage (learning rate, lr) de 0.00025. γ qui représente le décompte du reward au fil des itérations est initialisé à 0.95, proche de 1 pour que les récompenses suivantes aient une importance similaire à la récompense immédiate, l'objectif étant de maximiser la récompense à long-terme. La taille maximale de la mémoire est limitée à 10000 éléments et le facteur ϵ qui attribue une probabilité de choisir une action aléatoire est initialisé à 1 et décroit en utilisant la formule $\epsilon = \max(\epsilon_{\min}, \epsilon \times \epsilon_{\text{factor}})$ avec $\epsilon_{\min} = 0.07$ et $\epsilon_{\text{factor}} = 0.95$ ce qui assure la décroissance jusqu'à une valeur minimum d'aléatoire.

Un point important qui intervient dans les algorithmes de Q-Learning est le calcul de la récompense associée à une action. Afin de forcer l'agent à se rapprocher de la cible, on lui fournit une récompense égale à $-2 \cdot \sum_{i=1}^{n} \left(\frac{(\text{objectif}_i - \text{agent}_i.\text{position})}{\text{taille de la carte}} \right)^2$), il s'agit d'une distance euclidienne normalisée par la taille de la carte. Si l'agent se trouve sur une case rouge on soustrait 10 au reward et s'il atteint l'objectif on ajoute 100.

Conclusion et perspectives

Cette première expérience a permis d'explorer les bases du Deep Renforcement Learning mais elle présente de nombreux défauts. Bien que l'algorithme d'apprentissage converge et que l'agent atteigne la cible la majorité du temps en moins de 600 itérations en suivant le chemin vert, ce qu'il a appris n'est pas à reconnaître l'objectif et à s'en approcher mais simplement une succession de déplacement qui lui permettent de maximiser sa récompense. Preuve en est lorsqu'on conserve la même carte en déplaçant l'objectif, l'agent suit exactement le même parcours en se dirigeant vers l'emplacement précédent de la cible sans considération pour la nouvelle. Afin de palier à ce problème, il est nécessaire de générer aléatoirement des cartes lors de l'apprentissage.

Le modèle convergeait en un nombre d'époque raisonnable mais le temps d'un épisode semblait trop long à M. Pasquet alors des solutions d'optimisation du code ont été envisagée : privilégier le calcul matriciel au calcul séquentiel de certaines boucles "for", remplacer les "if" par des multiplications avec $1 -$ booléen et passer certaines opérations de traitements d'images sur le GPU. Pour justifier le passage des opérations sur le GPU, on peut comparer les temps de transfert et le nombre de bits transmis.

Ainsi lorsqu'on encodait la carte sur le CPU on multipliait presque par 10 le temps de transfert entre le CPU et le GPU. Cela s'avérait particulièrement coûteux dans notre cas où on sollicite régulièrement le réseau avec un batch de taille 128.

Comme a pu le suggérer la présentation de l'environnement, nous sommes loin d'une modélisation continue car l'agent évolue sur des cases de manière discrète. Il s'agit donc

	400x400x6 (float32)	**400x400x1 (float32)**	**400x400x1 (int8)**
Nombre de bits	30720000	5120000	1280000
Temps de transfert pour 1 élément	0.233542	0.234201	0.212489
Temps de transfert pour 128 éléments	0.017817	0.009304	0.001877

TABLE 2.1 – Tableau comparatif des temps de transferts entre CPU et GPU

d'une voie d'amélioration que nous aborderons dans les prochains algorithmes.

2.3.2 L'amélioration du Q-Learning

Cette partie décrit l'ensemble des améliorations et modifications sur l'algorithme de Q-Learning, l'objectif est de lister les problèmes rencontrés et la manière de les résoudre.

Modifications algorithmiques

Dans la nouvelle version de l'algorithme, une fonction génère des cartes aléatoires en utilisant des courbes de bézier qui permettent d'obtenir des tracé relativement lisses. La carte change donc à chaque épisode ce qui permet de s'assurer que l'agent n'apprenent pas un parcours par coeur. De plus les cartes sont réduites d'un facteur 10 par rapport à la première expérience, on essaiera de faire converger un agent sur une carte de 40×40. Dans un second temps, nous avons décidé de supprimer la double sortie du réseau car la façon dont les Q-valeurs étaient gérées pour la mise à jour des poids était incertaine. La sortie du réseau ne consiste donc qu'en une couche Dense et on fait correspondre un couple angle-vitesse à chaque sortie. Parallèlement à cela, une approche "semi-continue" a été envisagée avec une modification des actions : l'agent ne choisit pas une vitesse mais choisit d'augmenter, diminuer ou laisser constante celle-ci. Cela fonctionne par des additions ou soustraction de 0.05 clippé entre -1 et 5. Le même principe s'applique pour l'angle avec un ajout de $+/-18$ degrés, ce qui est équivalent en radians à ajouter $\frac{\pi}{10}$ à l'angle du robot. Les combinaisons de prédiction du réseau sont donc diminuées à 9 couples angle-vitesse. Le réseau a subi une autre modification, voir A.1, consistant globalement à réduire la taille des kernels de convolutions 2D et les strides pour conserver plus d'informations.

D'autres modifications permettant de rendre le code plus ergonomique ont également été appliquée avec notamment la suppression de la classe Robot. Ses attributs deviennent ceux de l'environnement et les hyper-paramètres qu'ils stockaient sont des constantes dans le script d'entrainement.

Résultats et interprétations

Lors des épisodes d'entrainement, l'agent ne parvient pas à trouver une solution systématique. Il atteint parfois l'objectif mais il possède un marge de pas généreuse pas épisode ce qui lui permet d'explorer beaucoup de cases. Pourtant il ne trouve pas toujours l'objectif et lorsqu'on observe son comportement en inférence on peut observer un phénomène intéressant. A défaut de comprendre qu'il doit atteindre le cercle blanc, il apprend que faire des cercles lui permet d'augmenter sa récompense. En effet, le reward étant défini comme une fonction de la distance, en décrivant des cercles il parvient à l'augmenter au moins pendant la moitié de ses actions. Ce comportement se généralise à toutes les instances de cartes aléatoires hors ce n'est pas ce qu'on attend de lui. Une redéfinition de la fonction de reward s'impose donc, la modification est inspirée de Chen et al. [20]. On définit une récompense de -0.1 sur l'ensemble des cases de la carte sauf lors de l'atteinte de l'objectif qui aura pour valeur 10. Les cases rouges n'infligent plus une pénalité directement sur le reward mais divisent la vitesse de l'agent par 4 ce qui devrait le forcer à y passer le moins de temps pour maximiser au plus vite sa récompense.

Optimisations de l'algorithme

Malgré cette redéfinition justifié de la récompense, le réseau ne converge pas et l'agent ne parvient pas à trouver la cible. A cela on peut poser trois hypothèses :

— Si l'agent ne parvient jamais a atteindre l'objectif alors il ne peut pas distinguer une bonne action d'une mauvaise car elles apportent toute la même récompense.
— Les récompenses positives étant rencontrées très rarement elles sont faiblement rejouées et ne contribuent pas à mettre à jour les poids.
— L'exploration de l'agent est insuffisante et il tend à rester dans les mêmes comportements, sans s'éloigner de son point de départ.

Chacune de ces hypothèses est une problématique à résoudre et nous allons les aborder individuellement dans un premier temps.

La solution pour que l'agent découvre des comportements récompensés est de lui fournir un oracle, c'est à dire de lui fournir des expériences qui sont assurément bonnes et mènent à la solution. C'est une pratique courante dans les jeux 3D impliquant du continu, on fait appel à expert/un joueur professionnel qui simule des parties pour l'agent de RL. A defaut de jouer directement la partie pour l'agent, on définit une probabilité que l'oracle prenne la main sur l'épisode et celui-ci choisit l'action qui permet d'aller dans la direction de la cible avec une vitesse adéquate selon la distance. C'est plutôt simple à coder lorsqu'on connaît les coordonnées de l'objectif et de l'agent mais ce n'est pas une solution optimale car on ne tient pas compte des cases rouges, une version améliorée qui a été évoquée serait d'utiliser une forme d'algorithme de Dijkstra pour définir le chemin optimal, on pourrait même laisser l'agent jouer et lui attribuer des récompenses plus importantes s'il suit le chemin optimal. Bien que cette alternative soit intéressantes elle n'a pas été

implémentée.

La problématique des récompenses rares a déjà été évoquée dans la revue de la littérature. Pour contrer ce phénomène on implémente une variante du replay d'expérience priorisé dans laquelle on définit le poids des transitions atteignant l'objectif à 10, les autres à 1. Les probabilités de tirages sont alors calculés comme énoncé en section 2.2 en divisant par la somme des poids. Cette solution a déjà fait ses preuves dans les travaux de Schaul et al. [10] mais nécessitent tout de même que l'agent atteigne l'objectif.

Afin que l'agent explore de nouvelles cases de la carte, on crée une matrice de la taille de la carte qui contient des 0 pour les cases non explorées et des 1 pour celles visitées. On la donne à l'agent en entrée du réseau, ce qui implique une légère modification de l'architecture, et sa récompense lorsqu'il re-visite une case est égale à -0.2. Il est donc davantage pénalisé et cela se remarque dans son comportement, pour éviter ces mauvaises récompenses il décrit des spirales depuis sa position initiale en augmentant sa vitesse. Il ne s'intéresse toujours pas à l'objectif mais cela lui permet d'explorer bien plus de cases.

La conclusion de ces trois solutions est qu'elles peuvent s'avérer insuffisantes si prises indépendamment. Nous les avons alors combinées dans un seul algorithme et les résultats se sont avérés bien meilleurs. Deux stratégies complémentaires proposées par M. Pasquet ont également été appliquées : ne pas changer la carte tant que l'agent n'est pas parvenu à trouver la solution et diminuer le learning rate de 20% au bout d'un nombre donné d'épisodes. Avec cette configuration, mon modèle tend à atteindre l'objectif dans 66% des cas en phase d'inférence. Il convient alors de se demander pourquoi le modèle qui possède l'ensemble des informations nécessaires ne parvient pas à trouver l'objectif. C'est l'expérience suivante qui a permis d'obtenir l'intuition qu'un problème indépendant du réseau en était responsable. Au travers des couches de convolution, l'agent va perdre sa position et ne plus être capable de prendre une bonne décision s'il ignore où il est placé. Une solution qui aurait pu être envisagée serait de réintroduire sa position au niveau du Flatten mais les nouvelles contraintes qui ont guidées la section suivante permettent de s'affranchir de ce problème.

2.3.3 Changement de perspective

Avant de pouvoir chercher à améliorer le modèle, il m'a été demandé d'effectuer une modification conceptuelle dans l'algorithme. Afin de se rapprocher de la réalité terrain, le robot ne reçoit plus une image de la carte en entrée mais un vecteur contenant l'ensemble des cases situées dans un cône devant lui. Cela a pour but d'imiter le fonctionnement d'un LiDAR (même si l'agent n'obtient pas la distance aux obstacles comme donnerait un LiDAR, d'autant plus qu'il n'y a pas d'obstacles), voir figure 2.4 . Cela implique une restructuration du modèle qui ne reçoit plus en entrée une image mais le vecteur sur lequel on effectue des convolutions 1D. On fournit également en information complémentaire la vitesse et l'orientation du robot ainsi que la direction de l'objectif, ce qui est une grosse information. Cependant ce n'est pas grâce à cette information que le modèle marche

mieux qu'avec l'image en Input. Comme mentionné auparavant, c'est le fait de pouvoir se replacer dans l'environnement qui permet à l'agent de prendre la meilleur décision. Dans le cadre de l'observation cônique il n'a plus besoin de savoir sa position car l'ensemble des informations y sont relatives.

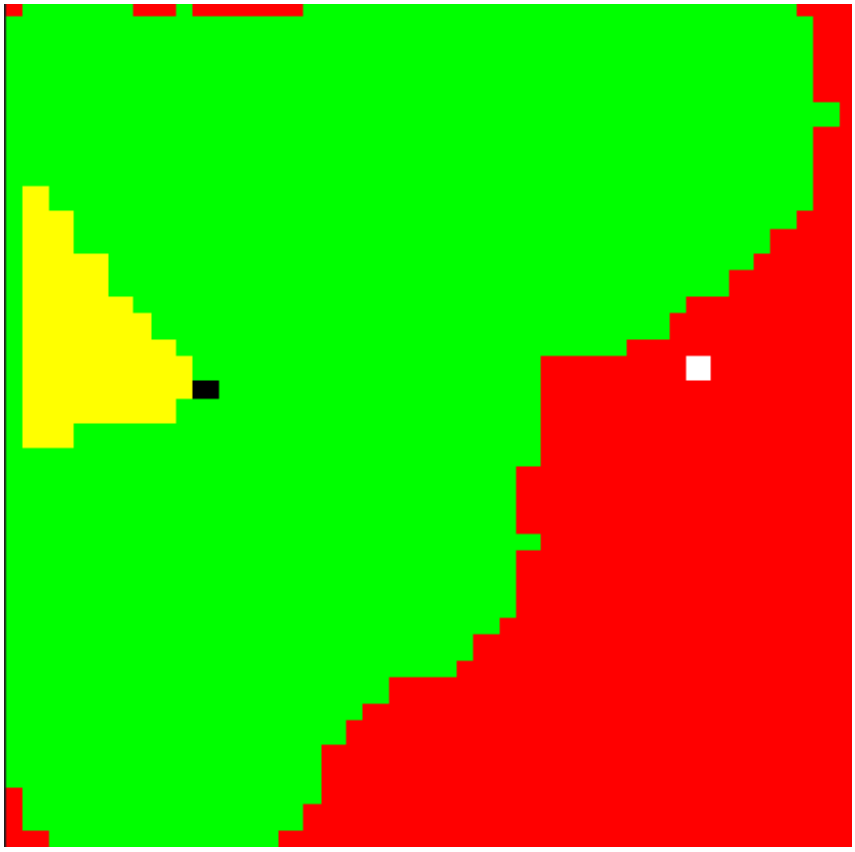


FIGURE 2.4 – Exemple d'une carte générée aléatoirement avec un agent qui reçoit une information par cônique

On commence l'entrainement sur une carte de 20×20 de laquelle l'agent reçoit un vecteur de 209 éléments (on considère le fonctionnement d'un LiDAR comme une liste uniforme d'angles qui "ping" une case à une distance variable dans un cône de 60 degrés alors une case peut apparaître plusieurs fois dans le vecteur). Le principe de *one-hot encoding* s'applique toujours et on ajoute des dimensions pour y insérer la vitesse, l'orientation et la direction de l'objectif, voir .

Les 209 éléments du vecteurs sont obtenus par la multiplication de 11 angles uniformément répartis entre orientation de l'agent $-\frac{\pi}{6}$ et orientation de l'agent $+\frac{\pi}{6}+\frac{\pi}{10}$ avec 19 distances de 1 à 19. Une deuxième version de l'observation a été conçu pour mieux s'adapter aux grandes cartes, on considère maintenant 21 angles et 39 distances pour obtenir un vecteur de 819 cses. Pour palier l'augmentation d'informations qui risquerait de devenir ininterprétable par le réseau après le Flatten, le nombre de neurones a été augmenté à 1024 sur les deux couches cachées.

Comparaison des modèles

Pour comparer des modèles de Deep Reinforcement Learning, plusieurs facteurs peuvent être pris en compte. D'une part la récompense moyenne au cours d'un épisode, le nombre de pas pour atteindre l'objectif et la vitesse de convergence. Ici on s'intéressera au capacités du modèle précédent à converger et atteindre son objectif sur des cartes de différentes tailles. Cela nous permettra de mettre en évidence deux stratégies d'apprentissage et leurs impacts : les réseaux *main* et *target* ainsi que la réutilisation de modèles déjà entrainés.

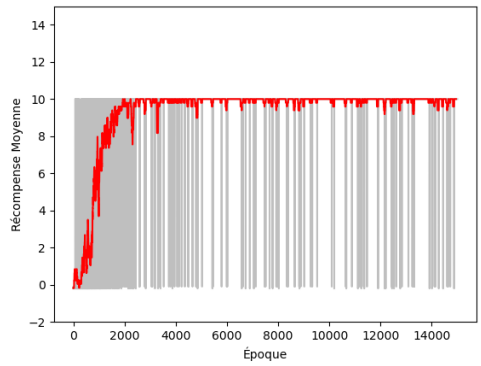
Un moyen d'assurer la convergence des modèles consiste à les entrainer sur des petites cartes puis d'augmenter progressivement la taille de la carte. Pour ce faire, on diminue le facteur aléatoire ϵ à 0.2 lors du nouvel entrainement afin que l'agent ne remplisse pas sa mémoire de trop d'exploration. On suppose que sa politique lui permet déjà d'atteindre l'objectif et que le réseau a simplement besoin d'être *fine-tuner*. Cela se se confirme plutôt bien sur la figure 2.5 dans laquelle on voit le premier modèle qui met environ 2000 itérations à ajuster ses poids pour maximiser sa récompense sur une carte de taille 40×40. Le deuxième modèle qui récupère les poids du premier est sujet à quelques instabilités mais il n'y a pas une phase de croissance comme dans le premier, il sait plus rapidement comment atteindre l'objectif et se stabilise au fil des itérations.

Bien que les modèles convergent de manière très nette dans le cas des cartes 4040, on pourrait reprocher une trop grande variance de la récompense ou du nombre de pas sur les cartes 100×100. Evidemment, il y a l'aléa de la génération des cartes qui entre en jeu mais les performances du modèles semblent surtout diminuer lorsqu'on regarde le reward.

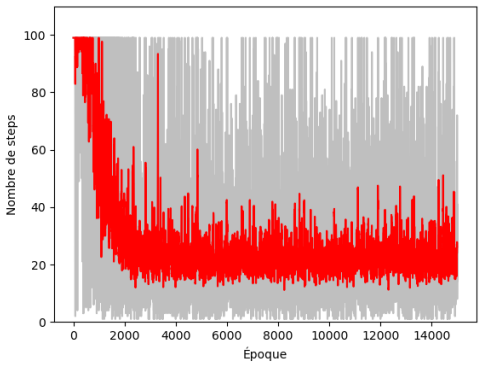
Les modèles *main* et *target* sont un moyen d'introduire de la stabilité comme expliqué dans la revue de la littérature. Cela consiste à créer deux modèles initialisés avec les mêmes poids, le réseau principal fait les prédictions pour les actions et le réseau target est mis à jour lors de l'expérience replay puis les poids sont synchronisé à intervalle régulier.

Pour comparer ces deux approches, on part d'un modèle entrainé sur du 40×40 et on observe les résultats sur des cartes de 100×100. Ils sont rapporté dans la figure 2.6 ci-dessous.

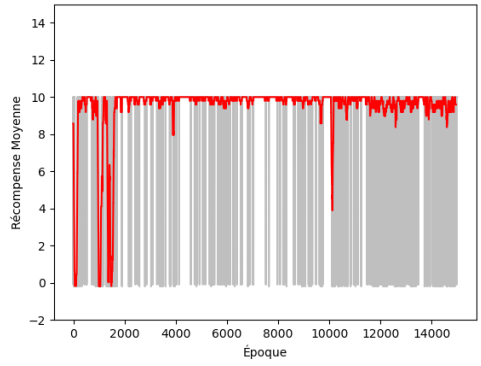
A première vue on pourrait penser que les résultats des modèles censés être plus stables ne le sont pas, notamment dans la variance des courbes de nombres de pas en rouge. Cependant il faut prendre en compte que le réseau qui fait la prédiction n'est pas mis à jour aussi souvent que dans l'expérience précédente ce qui veut dire que lorsque l'agent rencontre une carte "difficile" il reçoit moins de retour d'expérience et tend à rester bloqué plus longtemps. D'autant plus que les environnements ne changent pas tant que l'agent n'a pas trouvé la solution. Pour contrer cela, les poids sont synchronisés toutes les 100 itérations. Une fois que l'on a intégré cette information, on peut observer que la récompense du modèle avec les réseau *main* et *target* est beaucoup plus stable au delà de 10000 épisodes. En phase d'inférence, on peut voir que les agents semblent suivre des comportements vraiment intelligents en privilégiant les chemins verts aux zones rouges si possible, des gifs sont disponible sur le dépôt github StageRL-MTD.



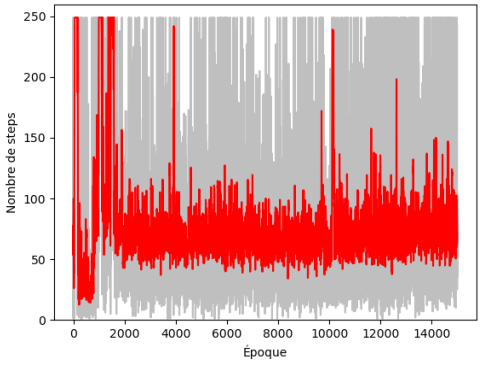
(a) Récompense carte 40x40



(b) Nombre de pas carte 40x40

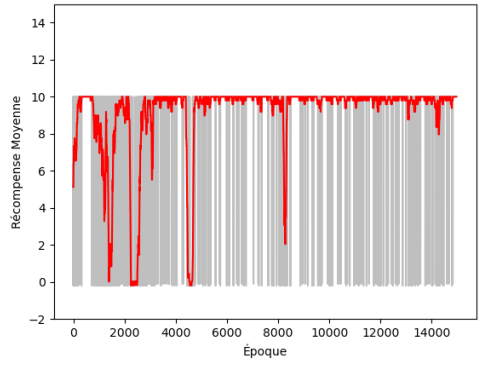


(c) Récompense carte 100x100

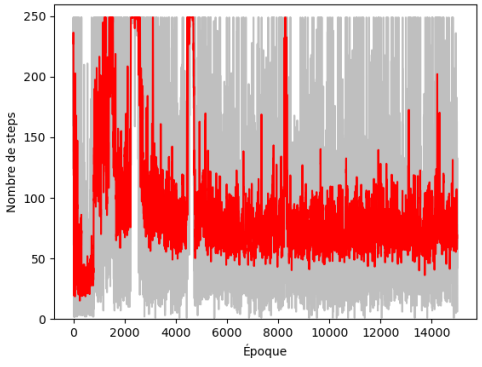


(d) Nombre de pas carte 100x100

FIGURE 2.5 – Convergence du réseau avec en (a) la récompense et (b) le nombre de pas le réseau entrainé "from scratch". En (c) et (d) les mêmes métriques pour le réseau entrainé avec les poids de rechargé après la convergence. En rouge une moyenne glissante sur 10 épisodes, en gris les valeurs par épisode.



(a) Récompense réseau "stable"

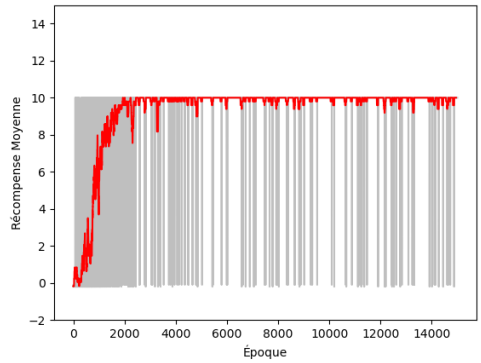


(b) Nombre de pas réseau "stable"

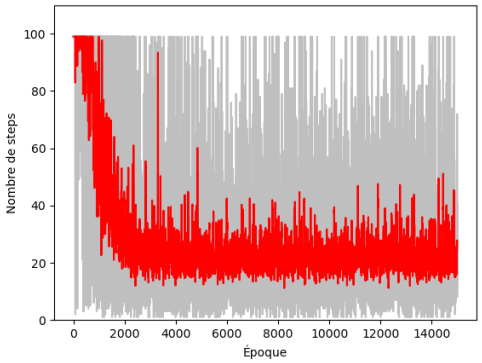
FIGURE 2.6 – Convergence des modèles stabilisés par le système de réseau principal et réseau cible

Une fois ces deux modèles entrainés sur des cartes de 100×100, l'objectif est de continuer à agrandir l'environnement avec des cartes de 200×200. On va également essayer de comparer les deux approches précédentes. Comme l'augmentation de la taille étant plutôt grande, j'ai implémenté un apprentissage "doux" tel que les objectifs sont générés à une distance croissante de l'agent. Pendant les 8000 premiers épisodes, l'objectif se trouve à une distance aléatoire comprise entre 0 et $3 + \frac{\text{nb}_{\text{episodes}}+1}{40}$ permettant une augmentation croissante de la difficulté. Le nombre maximal d'itération par épisode est défini à 500 ce qui est largement suffisant étant donné que la distance maximale entre deux points est de 282.

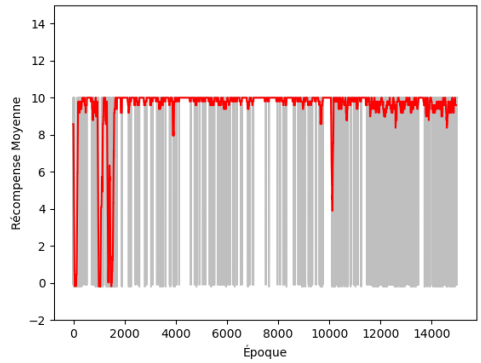
CHANGER LES GRAPHIQUES AVEC CEUX EN 200 PAR 200 (EN TRAIN DE TOURNER)



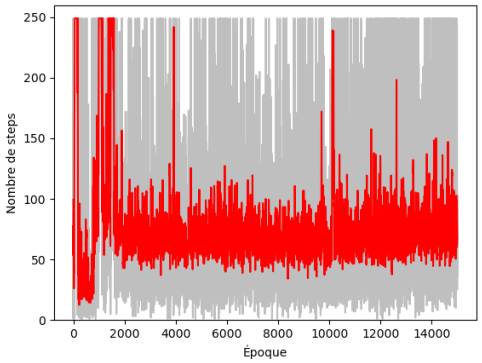
(a) Récompense réseau seul



(b) Nombre de pas réseau seul



(c) Récompense carte main/target



(d) Nombre de pas main/target

FIGURE 2.7 – Convergence du réseau simple (a) et (b) vs les réseaux main/target (c) et (d) sur des cartes 200×200

On constate cependant que les réseaux divergent totalement après plusieurs milliers d'épisodes. On peut supposer à cela que la distance est trop grande pour que l'agent et qu'il aurait fallu opter pour un augmentation un peu moins violente de la taille de la carte. La difficulté croit peut-être également de manière trop lente car le réseau simple diverge à partir des 8000 épisodes, là ou l'aléa classique prend le relai. Il serait intéressant

d'effectuer les tests pour voir les différences mais l'entrainement des modèles en 200×200 prend plus d'une dizaine d'heures pour effectuer les 15000 épisodes. Des stratégies de parallélisations pourraient être intéressante à implémenter.

Ajout d'obstacles

Après avoir réussi à entraîner un agent sur notre monde simple, il fallait lui ajouter de la complexité. Des obstacles matérialisés par des disques de rayon 5 sont ajoutés sur la carte. Il faut ajouter une dimension au *one-hot encoding* du vecteur d'entrée mais sinon le réseau est globalement le même. L'obstacle a pour effet de tuer l'agent mettant fin à l'épisode et lui attribue une récompense de -10. Le réseau est entrainé dans un premier temps sur une carte de 40×40 que l'on tente d'étendre à 80×80 en reprenant les mêmes poids. Un phénomène a été observé, parfois l'agent semble apparaître juste devant l'objectif car il meurt instantannément. Comme les cartes ne sont pas censé changer avant la complétition, on introduit ici une petite variante dans laquelle si l'agent échoue 10 fois consécutive alors la carte change.

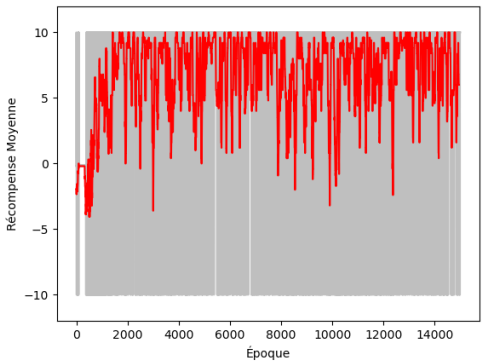
La convergence du réseau est bien moins marquée que dans l'expérience précédente, cependant en considérant les problématiques d'initialisations qui n'ont pas été très bien gérées, on retrouve une forme de convergence avec un reward au dessus de la moyenne. Cet entrainement n'a été effectué qu'avec un obstacle placé aléatoirement vers le centre de la carte ce qui veut dire qu'il ne représente pas une grosse gêne pour l'agent.

La métrique du nombre de pas perd un peu de son sens car une mort de l'agent entraine la fin de l'épisode mais ne signifie pas pour autant un succès. Les résultats de récompenses sur les cartes de 80×80 sont moins bons et on peut supposer à cela que si le modèle chargé n'est déjà pas optimal alors une bonne convergence n'est pas assurée. Un modèle s'entraine à l'heure actuelle sur une carte de 40×40 avec deux obstacles pour essayer de mieux les gérer, et ce sur un plus grand nombre d'épisodes.

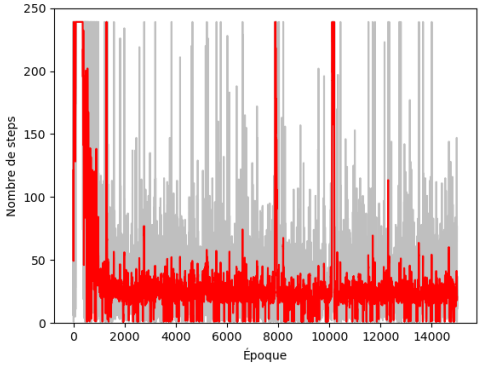
Une fois que l'agent parviendra à reconnaître les obstacles, on pourra lui simuler un circuit (entouré d'obstacles et avec un au milieu) en faisant apparaitre des objectifs intermédiaires sur la carte. Cela reviendra à réaliser des tours de pistes comme le souhaite notre problématique initiale.

2.3.4 Essai au gradient de politique

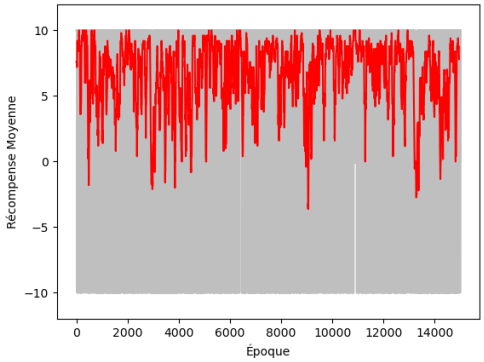
Le gradient de politique est la deuxième grande famille d'algorithmes, son avantage principal étant d'estimer les probabilités de chaque action sans avoir à bien estimer les valeurs de la fonction $Q(s, a)$. C'est également la solution privilégiée dans les problèmes continus en la combinant avec une estimation de la fonction $V(s)$ par Q-Learning : c'est le principe d'acteur-critique. Hors notre problématique de recherche n'est pas seulement de faire de la régression mais de faire de la régression avec du Q-Learning, cela se fait très peu actuellement et on espère faire mieux.



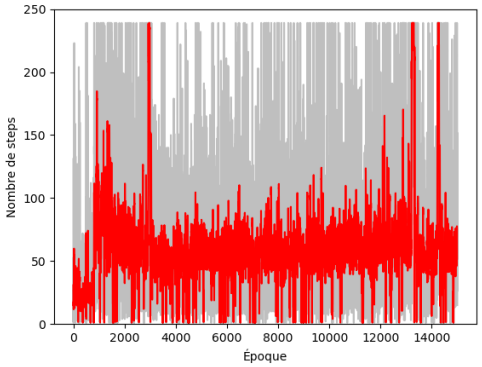
(a) Récompense 40x40



(b) Nombre de pas 40x40



(c) Récompense 80x80



(d) Nombre de pas 80x80

FIGURE 2.8 – Convergence "from scratch" du modèle sur des cartes de taille 40×40 fig (a) et (b) puis sur des cartes 80×80 fig (c) et (d) en récupérant les poids

Développer un modèle acteur-critique utilisant le système de gradient de politique n'est cependant pas dénué de sens. Cela étant l'état de l'art sur les problèmes de régression, il est essentiel de comparer notre futur modèle de Q-Learning avec cette *baseline*.

Le stage ne s'étant pas orienté dans cette direction pour l'instant, j'ai juste essayé de reproduire l'implémentatio Keras du DDPG et du PPO [13], [21]

2.4 Perspectives

A EXPLIQUER LA SUITE : ACTEUR CRITIQUE GP POUR COMPARAISON AVEC REGRESSION QLEARNING + GERER LES MULTI AGENTS : PROBLEMES DE DISTINGUER LES INPUTS + PLUSIEURS CONFIGURATIONS POSSIBLES TOUT EN RESPECTANT RATIO DE DISTANCES

2.5 Conclusion

Conclusion

Au cours de cette première année de Master MIASHS, j'ai eu l'occasion d'aborder la science des données sous deux de ses aspects. Tout d'abord dans le contexte d'une entreprise où les compétences théoriques apprises en Master n'ont que peu leurs places. Cela m'a permit entre autre de me former aux outils d'analyse de données "grands publics" comme les outils de DataViz ou les ETL. Découvrir le cadre d'une entreprise s'est également avéré intéressant car on est souvent confronté aux volontés d'autres acteurs qui doivent co-exister et collaborer.

Dans un second temps, le stage de recherche à la Maison de la Télé-Détection a confirmé mon intérêt pour le monde de la recherche et mon ambition de poursuivre les études après le Master avec un doctorat. Lors de ces quelques semaines j'ai eu l'impression d'effectuer des missions en lien direct avec le Master MIASHS et de développer des compétences qui me seront directement utiles lors des cours de l'année prochaine. L'aspect réflexion du stage est particulièrement intéressant aussi, on produit un travail sur lequel on fournit une analyse et à laquelle on doit (éventuellement) trouver une solution. Cet aspect de la science des données est trop souvent négligé que ce soit dans les cours ou l'entreprise.

Lors du reste de mon stage, j'espère bien avoir l'occasion de découvrir de nouveaux concepts de Deep Learning et pouvoir les appliquer dans le cadre de la recherche.

Annexes

A Les différentes architectures

A.1 Premier réseau

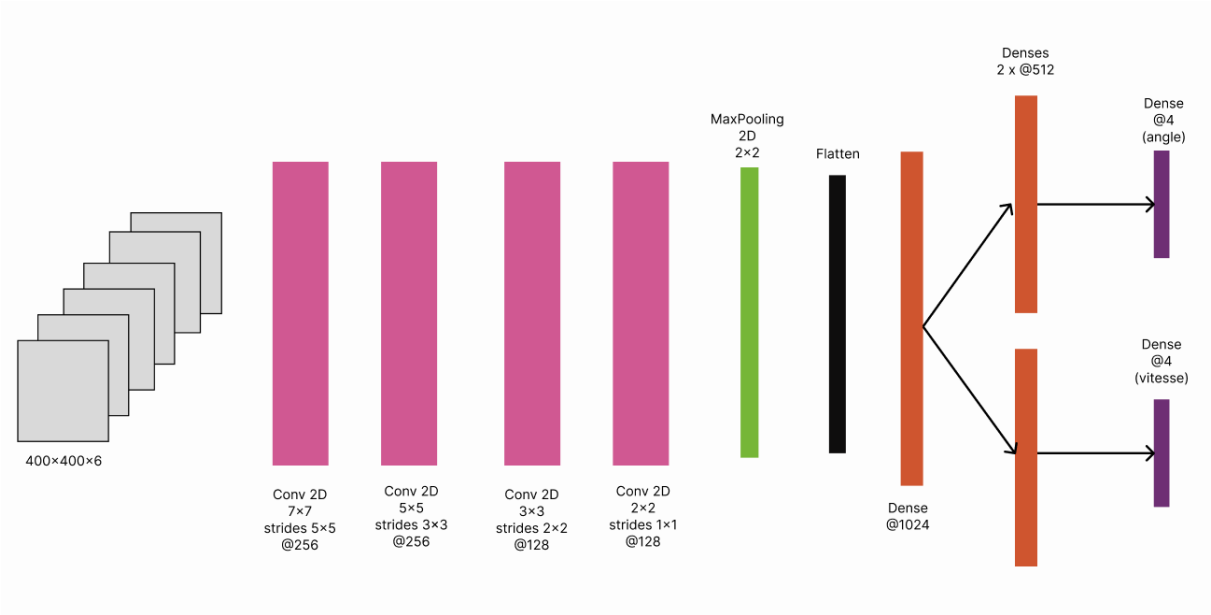


FIGURE A.1 – Premier réseau utilisé pour les expériences de Deep RL

Ce premier réseau possède deux sorties pour estimer deux valeurs Q pour deux actions différentes qui doivent agir de manière complémentaire pour estimer l'orientation et la vitesse de l'agent de RL. Toutes les fonctions d'activation sont des ReLU : $\text{ReLU}(x) = \max(0, x)$

A.2 Second réseau

Ce second réseau a l'avantage de conserver davantage d'informations que le premier, il n'estime qu'une seule valeur Q qui est mappé à une valeur d'angle et de vitesse. Il est conçu pour des observations de l'environnement de taille moindre que le premier.

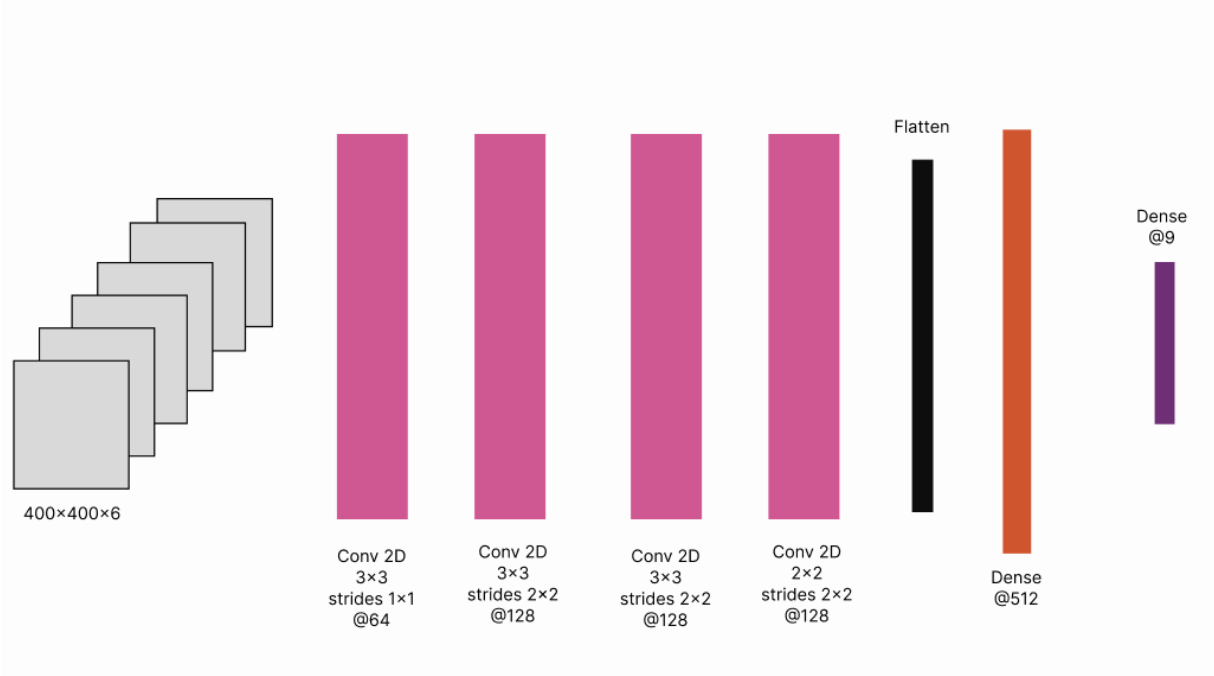


FIGURE A.2 – Second réseau utilisé pour les expériences de Deep RL

Bibliographie

[1] Alternant BI Enzo Dardaillon. *Présentation de l'architecture des serveurs BI*. Capelle, 2023.

[2] Groupe Capelle. Site du groupe capelle : Un transporteur expert des transports conventionnels et eceptionnels, 2024.

[3] Documentation pentaho, 2024.

[4] Documentation numpy, 2024.

[5] Documentation pandas, 2023.

[6] Documentation tableau, 2024.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv :1312.5602*, 2013. arXiv :1312.5602.

[8] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. 2015.

[9] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. 2015.

[10] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv :1511.05952*, 2016.

[11] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. 2015.

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv :1707.06347*, 2017.

[13] implémentation keras du ppo, 2024.

[14] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv :1509.02971*, 2015.

[15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv :1506.02438*, 2015.

[16] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv :1803.02811*, 2018.

[17] Daniel Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv :1803.00933*, 2018.

[18] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv :1602.01783*, 2016.

[19] Implémentation keras du q-learning, 2024.

[20] Hao-Yuan Chen, Yen-Jui Chang, Shih-Wei Liao, and Ching-Ray Chang. Deep-q learning with hybrid quantum neural network on solving maze problems. *Department of Computer Science, University of London, London WC1E 7HU, United Kingdom*, 2023.

[21] implémentation keras du ddpg, 2024.