

RAPPORT DE PSC - INF N°09

Étude sur les stratégies de navigation de ballons stratosphériques

2023-2024

Coordinateur : Eric GOUBAULT Tuteur : Hugo MARCHAND

Mohamed ALOULOU Moncef DJAFRI Tudy HERRIOU
Mathias PEREZ Hippolyte WALLAERT



INSTITUT
POLYTECHNIQUE
DE PARIS



RÉSUMÉ

Notre Projet Scientifique Collectif (PSC) [1] a pour objectif la production d'un algorithme permettant le **contrôle de la trajectoire d'un ballon stratosphérique** afin d'atteindre une destination donnée. Il répond à une demande de la **start-up Stratolia**, qui utilise des ballons stratosphériques pour fournir des images haute résolution de la surface terrestre. Stratolia espère pallier la limitation fréquentielle des acquisitions d'images par satellite. En effet, même les constellations de satellites les plus performantes sont contraintes à fournir seulement une à deux captures par jour pour une zone donnée. À l'opposé, Stratolia propose une capacité d'observation en temps réel, permettant des acquisitions d'images instantanées au moment précis où les événements surviennent.

La navigation et la préparation d'un itinéraire pour un ballon stratosphérique constituent des tâches complexes qui nécessitent une intégration et une synchronisation précises de diverses données et technologies. La difficulté principale réside dans le traitement et l'interprétation efficace d'une quantité massive d'informations sur les **vents stratosphériques**. En outre, ces données doivent être constamment mises à jour et ajustées en raison des erreurs potentielles dans les prédictions météorologiques.

Sous l'hypothèse initiale de **connaissance absolue des vents**, nous avons conçu et implémenté des **algorithmes dépourvus de techniques d'apprentissage automatique** pour résoudre le problème de planification d'itinéraires. Ces algorithmes, méticuleusement optimisés, présentent des performances très satisfaisantes ainsi que des avantages distincts, notamment en termes de fiabilité et de rapidité de traitement. Ils constituent une base solide, capable de fournir des solutions de navigation précises. Par ailleurs, afin de permettre une visualisation de nos résultats, nous avons développé toutes les dépendances graphiques d'affichage nécessaires.

Après une période approfondie de recherche et d'exploration des domaines du *Reinforcement Learning* et des réseaux de neurones, nous avons également progressé vers l'intégration de l'apprentissage automatique. En exploitant les ensembles de données accumulées par nos premiers algorithmes, nous avons développé des **agents intelligents capables de générer des itinéraires via l'apprentissage supervisé**. Cette étape a été suivie de l'adoption de l'apprentissage par renforcement, permettant ainsi une amélioration continue des stratégies d'itinéraire. Cette évolution méthodologique enrichit notre système de navigation, ouvrant ainsi la voie à l'intégration de **données bayésiennes sur les vents** et à **l'ajout de nouvelles contraintes sur le déplacement des ballons**.

Table des matières

1	Introduction	5
1.1	Présentation du sujet et contexte de l'étude	5
1.2	Approche initiale	6
2	Analyse des données et déplacement du ballon	7
2.1	Présentation et analyse des données de vent	7
2.2	Déplacement du ballon	8
2.2.1	Point de vue général	8
2.2.2	Déplacement du ballon au sein de nos algorithmes	10
3	Approche Directe avec Algorithmique Traditionnelle	11
3.1	Algorithme "naïf" : limite d'éloignement	13
3.1.1	Fonctionnement	13
3.1.2	Résultats	13
3.1.3	Bilan	14
3.2	Algorithme "sélectif" : parcours linéaire	15
3.2.1	Fonctionnement :	15
3.2.2	Résultats	15
3.2.3	Bilan	16
3.3	Algorithme glouton	16
3.3.1	Fonctionnement	16
3.3.2	Résultats	16
3.3.3	Bilan	17
3.4	Algorithme <i>wide-search</i> : maximisation de l'espace couvert	18
3.4.1	Résultats	19
3.4.2	Bilan	19
3.5	Synthèse sur la performance des algorithmes classiques	20
4	Approche par <i>Machine Learning</i>	21
4.1	Approche générale	21
4.1.1	Objectif	22

4.1.2	Les données	22
4.2	Apprentissage supervisé	23
4.2.1	Réseaux de Neurones Convolutifs	23
4.2.2	Les outils utilisés dans nos modèles	24
4.2.3	Les modèles	26
4.2.4	Méthode d'apprentissage	29
4.2.5	Implémentations et résultats	30
4.2.6	Bilan	34
4.3	Apprentissage par renforcement	34
4.3.1	Rappel sur le cadre conceptuel du <i>Reinforcement Learning</i>	34
4.3.2	Description de l'environnement et du système de récompense	35
4.3.3	Utilisation du <i>Deep Q-Learning</i>	36
4.3.4	Bilan et difficultés rencontrées	37
5	Conclusion	38
5.1	Résultat final	38
5.2	Pistes d'amélioration	38
Bibliographie		40
A Annexe : Majoration du nombre de points explorés dans l'algorithme <i>wide-search</i>		41

1

INTRODUCTION

1.1 PRÉSENTATION DU SUJET ET CONTEXTE DE L'ÉTUDE

Notre Projet Scientifique Collectif (PSC) [1] a pour objet la réalisation d'un agent permettant le contrôle de la trajectoire d'un ballon stratosphérique.

Ce projet est en partenariat avec **la start-up Satratolia** qui utilise des flottes de ballons d'hélium évoluant dans la stratosphère afin d'obtenir des photographies à haute résolution d'une zone donnée. Stratolia nous a proposé d'aborder la problématique du contrôle de la trajectoire de leurs ballons afin de s'assurer que les ballons atteignent la zone à observer.

La solution privilégiée par Stratolia est de piloter à distance l'altitude des ballons afin de s'appuyer sur **les différences de vitesse et de sens du vent selon l'altitude** pour se déplacer de la meilleure manière possible. La mise en pratique de cette idée requiert une implémentation algorithmique capable de fournir des instructions précises au ballon, basées sur des prévisions en temps réel de la direction du vent selon l'altitude.

Dès lors, l'enjeu principal de notre projet a été d'essayer d'apporter une réponse algorithmique à ce problème, via la recherche et la comparaison de plusieurs méthodes s'appuyant, ou non, sur le *Machine Learning*.

Le domaine de l'étude de la navigation des ballons stratosphériques est assez restreint. Certains projets ont essayé de développer des algorithmes pour contrôler des flottes de ballons, mais le domaine de recherche est loin d'avoir été complètement exploré. Le calcul physique de la trajectoire des ballons en fonction de la direction des vents est connu et ne sera pas abordé dans ce travail.

Face à cet enjeu, notre objectif final était de réaliser un agent permettant d'une part, en supposant connaître les vents futurs, de générer une liste d'instruction précise permettant de **guider le ballon d'un point A à un point B**, et d'autre part, en situation réelle face à des données de vents imprécises, d'adapter le comportement du ballon en direct.

Les principales contraintes de notre agent sont les suivantes :

- Vitesse du trajet prévu.
- Fiabilité de l'algorithme.
- Précision du trajet (la caméra embarquée sur les ballons propose un champ de vision de 40 kilomètres).

1.2 APPROCHE INITIALE

Nous avons décidé d'aborder le problème sous **deux perspectives distinctes**, chacune caractérisée par ses propres spécificités et avantages. Dans une première approche classique, nous avons décidé de concevoir un algorithme qui ne recourt pas à l'apprentissage automatique. L'ambition n'était pas de résoudre le problème directement de cette manière mais d'obtenir un algorithme fiable et interprétable. Simultanément, nous avons exploré une deuxième méthode exploitant les capacités du *Machine Learning*, en particulier du *Reinforcement Learning* et du *Supervised Learning*, pour élaborer un algorithme résultant d'un vaste ensemble de simulations.

Indépendamment de l'approche choisie, notre démarche initiale consistait à considérer que nous avions **une connaissance absolue sur les vents**, ce qui nous a permis d'établir une base de référence pour notre travail. Toutefois, nous avons gardé à l'esprit qu'il serait ensuite crucial de prendre en compte **les incertitudes inhérentes à tout modèle de prédiction des vents** et de les intégrer méticuleusement dans nos algorithmes.

Cette démarche progressive nous aura permis d'explorer l'ensemble du spectre des solutions, de la simplicité initiale à la sophistication des dernières approches, tout en garantissant que nos algorithmes soient capables de s'adapter à des scénarios réalistes. Finalement, notre objectif était de concevoir des solutions robustes et flexibles pour le déplacement précis et fiable de ballons stratosphériques, en tirant parti des avantages de l'approche naïve et des capacités puissantes du *Machine Learning*.

2

ANALYSE DES DONNÉES ET DÉPLACEMENT DU BALLON

Les données de vent disponibles jouent un rôle majeur dans la résolution du problème. En effet, tous nos algorithmes doivent inévitablement s'adapter aux données de vent disponibles et les exploiter au mieux pour obtenir des résultats satisfaisants.

2.1 PRÉSENTATION ET ANALYSE DES DONNÉES DE VENT

Les données de vent disponibles sont de deux sortes. Celles publiées par la NOAA (*National Oceanic and Atmospheric Administration*) et celles publiées par le EMCWF (*European Centre for Medium-Range Weather Forecasts*).

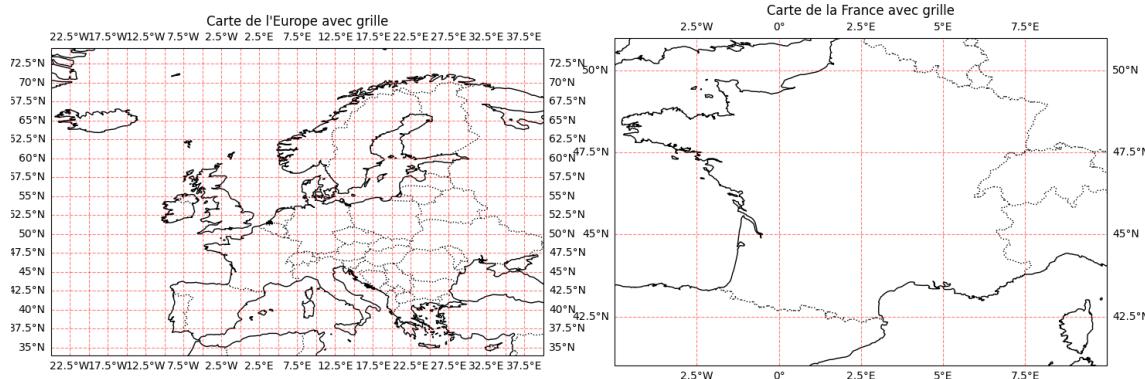


Figure 1: Cartes montrant les zones de collecte des données de vent par la NOAA.

Nous avons décidé de travailler avec **les données fournies par la NOAA** car elles étaient moins volumineuses et donc bien plus faciles d'utilisation. Ces données prennent la forme d'un vaste tableau regroupant les données de vent pour une année, avec une mesure toutes les six heures sur 10 512 points répartis sur la surface du globe. Ces points sont choisis tous les 2.5° de longitude et de latitude, ce qui mène précisément à 144 subdivisions de longitude et 73 de latitude. On dispose donc de 178 704 données de vents à un temps donné. Il faut noter que les données de vent disponibles sont particulièrement **grossières** et correspondent à **des grandes cases qui quadrillent la surface terrestre**. La hauteur des cases est de 278 km tandis que leur largeur (qui dépend de la latitude) vaut environ 190 km au niveau de la France.

Les vents sont décrits par un couple de coordonnées qui décrivent respectivement la vitesse du vent selon la direction Sud-Nord et la direction Ouest-Est. Concernant la distribution de ces données de vent, celles-ci ne sont pas identiques selon les deux directions et cela joue un rôle important dans l'analyse du problème.

Moyennes des vents

Sud-Nord : + 0.04 m/s
Ouest-Est : + 6.89 m/s

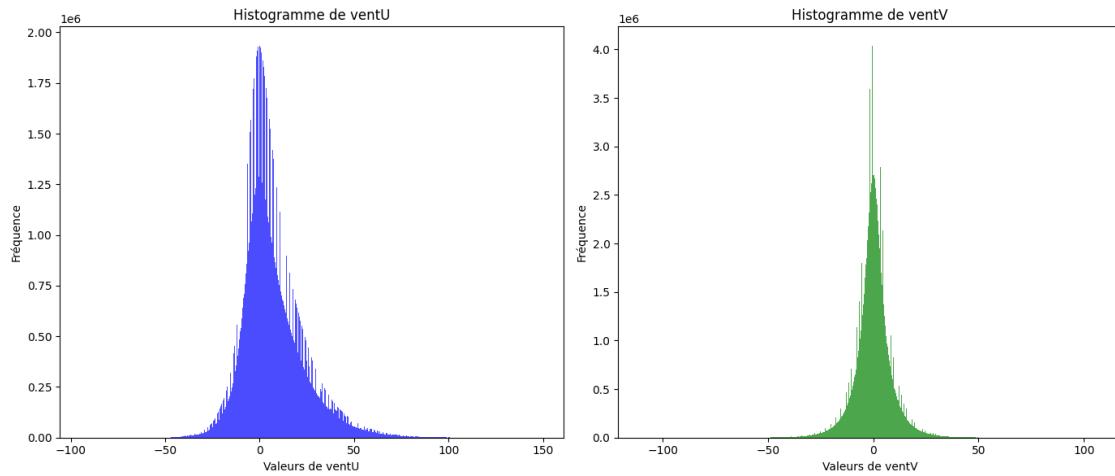


Figure 2: Histogrammes des vents Ouest-Est (ventU) et Sud-Nord (ventV)

On remarque que les vents sont symétriques¹ selon la **direction Sud-Nord** mais pas la **direction Ouest-Est**. Ceci n'est pas surprenant et est dû à la rotation de la Terre. Par contre cela a un impact important sur notre recherche de chemins entre deux villes données. En effet, en raison de la distribution particulière des vents, il est **plus facile de se déplacer d'ouest en est**, que dans le sens inverse. Ceci pourra expliquer notamment pourquoi nos algorithmes ne parviennent pas à trouver de chemin dans certains cas, particulièrement quand il s'agit d'un chemin d'est en ouest.

2.2 DÉPLACEMENT DU BALLON

2.2.1 • POINT DE VUE GÉNÉRAL

Avant toute chose, pour exploiter les données de vent de manière efficace et vérifier la cohérence des algorithmes que nous prévoyions de développer, nous avons programmé **une interface d'affichage graphique**. Cet outil nous a permis de visualiser les directions et les intensités des vents à différentes altitudes, ainsi qu'à différents intervalles de longitudes, latitudes et altitudes à un moment donné.

¹Plus précisément, une analyse des courbes montre que la courbe s'approxime bien par une gaussienne normale-inverse (NIG).

L'interface propose des représentations en 3D, en 2D, ainsi qu'en format GIF animé. Les visualisations nous permettent d'observer précisément les variations des vents.

Les ballons stratosphériques se déplacent en exploitant les vents à diverses altitudes de la stratosphère. En l'occurrence, **la direction horizontale des vents** est différente en fonction de l'altitude. Les ballons tirent alors parti de ces différences de directions en modifiant leur altitude, ce qui leur permet de suivre une trajectoire définie. Généralement, leur altitude est contrôlée en ajustant la quantité de gaz à l'intérieur de l'enveloppe, ce qui permet au ballon de **monter ou de descendre au besoin**.

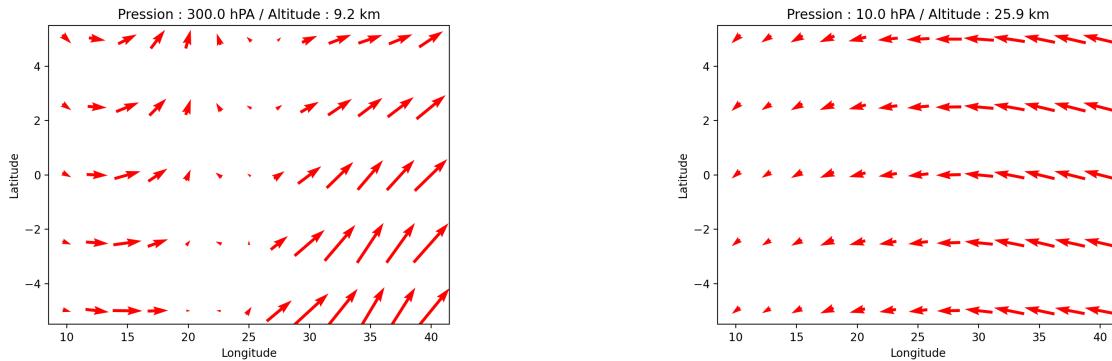
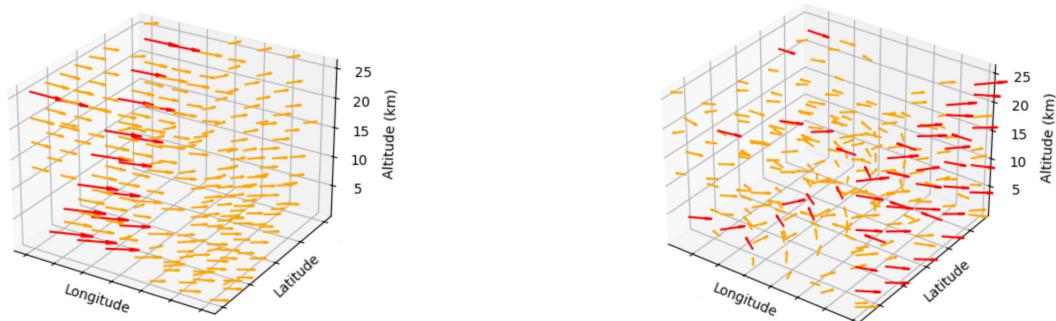


Figure 3: Comparaison simultanée des directions des vents à deux altitudes différentes.

La direction du vent exerce ainsi un impact déterminant sur la navigation des ballons. Or, les données de vent disponibles sont généralement imprécises et peu exploitées. Ces incertitudes quant aux données initiales sont au cœur du problème car elles constituent un obstacle face auquel les algorithmes se doivent d'être robustes. Ainsi, la compréhension fine des modèles de vent en altitude est cruciale pour maximiser l'efficacité opérationnelle des ballons stratosphériques.



(a) Vents propices au déplacement selon une unique direction.

(b) Vents favorables pour bien naviguer : les directions sont différentes.

Figure 4: Comparaison entre deux cartes de vents.

2.2.2 • DÉPLACEMENT DU BALLON AU SEIN DE NOS ALGORITHMES

Tous les algorithmes que nous avons développés s'appuient sur une fonction de déplacement du ballon appelée **parcours** qui implémente le déplacement du ballon à altitude constante. Au cours de nos travaux, nous en avons développé deux versions : parcours et parcours_{interpolate}. Les deux fonctions prennent en entrée **un noeud** qui correspondra à ce qu'on appelle un point tout au long de ce rapport, c'est-à-dire une longitude, une latitude, un niveau de pression, un temps et un père (pour pouvoir retracer le chemin parcouru une fois la solution trouvée). Les autres paramètres importants sont le temps de dérive $t_{\text{dérive}}$ qui fixe la durée de déplacement du ballon, ainsi que la précision voulue vis-à-vis de la destination, en dessous de laquelle on considère l'avoir atteinte. Les deux fonctions renvoient ensuite un booléen indiquant si la destination a été rencontrée au cours du parcours ainsi que le nouveau noeud correspondant à la nouvelle situation du ballon.

Concernant leur fonctionnement en tant que tel, les deux fonctions vérifient à intervalle constant² noté **temps_{test arrivée}** si la destination est atteinte et mettent à jour les valeurs du vent. La différence entre les deux fonctions réside précisément dans cette mise à jour :

- La première version se contente de récupérer les données de vent dans la case où se situe le ballon.
- La deuxième version part du constat que les données de vent restent assez imprécises et qu'il est légitime de vouloir les raffiner. La méthode adoptée est d'**interpoler linéairement les données de vent** dans toutes les cases qui entourent le point. Formellement, chaque point est situé au sein d'une case définie par quatre sommets (où les données de vent sont relevées). L'interpolation calcule une moyenne pondérée (par la distance du point aux sommets) des 8 points qui l'entourent dans l'espace longitude × latitude × temps.

Bien que plus simple en apparence, la première version implique de nombreuses disjonctions de cas lors des franchissements de cases. En effet, il faut reconstruire de nouvelles valeurs de vent et donc en amont avoir anticipé ce changement de case.

Des comportements pathologiques surviennent tout particulièrement dès lors qu'il y a un franchissement de cases à répétition, ce qui est le cas lorsque deux vents contraires se font face ou plus simplement lorsqu'on s'approche des pôles où les cases sont de plus en plus fines.

À l'inverse, la deuxième version ne souffre pas de ces problèmes car l'interpolation rend superflue la nécessité de mettre à jour les vents dès qu'il y a un changement de case. En effet les valeurs des vents sont désormais **continues** et donc il est suffisant d'attendre le prochain **temps_{test arrivée}** pour mettre à jour les vents.

Logiquement, nous avons donc décidé de travailler avec la fonction parcours_{interpolate} dans chacun de nos algorithmes, ceux-ci développant cependant des approches différentes face au problème.

²Ce temps est choisi en fonction de la précision et de la valeur moyenne des vents et vaut un peu plus de 20 minutes pour une précision de 10 km.

3

APPROCHE DIRECTE AVEC ALGORITHMIQUE TRADITIONNELLE

La première partie du problème est de parvenir à trouver des chemins entre deux points différents, sans prendre en compte les incertitudes liées aux données de vent. Ainsi, dans toute cette partie on fera **l'hypothèse que les données de vent exploitées sont exactes**. En tout cas on ne se souciera pas de vérifier que nos algorithmes sont robustes face aux imprécisions des données de vent. En l'occurrence ils ne le sont pas comme cela sera expliqué à la fin de la partie.

Pour résoudre ce problème de navigation exacte, nous avons ainsi développé au cours de notre étude plusieurs algorithmes aux fonctionnements différents et aux avantages et inconvénients distincts, chacun répondant à certains des enjeux de notre problématique initiale. Néanmoins ces algorithmes reposent sur un cadre d'étude commun qui permet, entre autres, de pouvoir comparer leurs performances. C'est l'objet de ce qui va suivre.

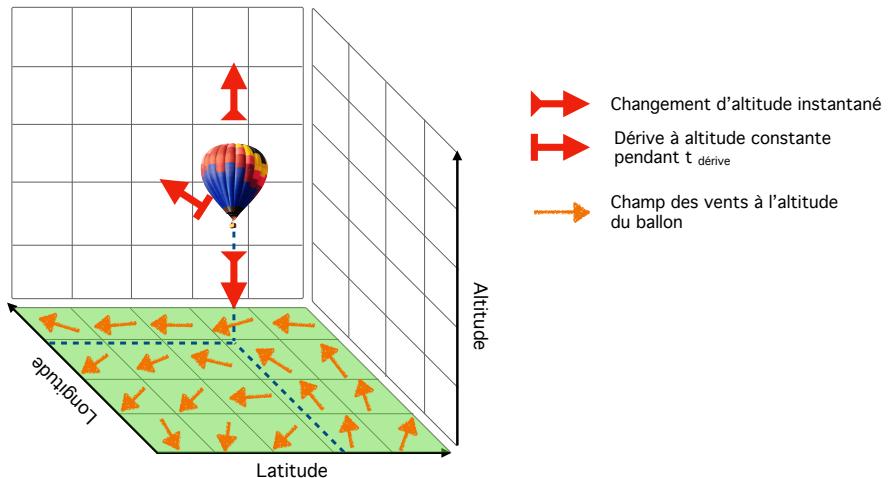


Figure 5: Comportements possibles du ballon

Tout d'abord, nous considérons qu'il y a **3 actions possibles** pour un ballon : monter en altitude, descendre en altitude, se faire porter par les vents à altitude constante (ce qu'on nommera la dérive). Ensuite, chacun de ces algorithmes suit l'hypothèse majeure suivante : **un changement d'altitude se fait instantanément**. En effet, lors de nos échanges avec Stratolia, nous avons recueilli des données de vitesses de ballon et il est apparu qu'un ballon pouvait aller du sol à une altitude de 22 000 m en une trentaine de minutes seulement. Étant donné que nos données de vents ne dépassent pas

cette altitude limite et que la résolution temporelle de nos données est d'un point toutes les 6 heures, cette hypothèse nous a semblée cohérente et justifiée. Ensuite, nos algorithmes prennent en compte plusieurs paramètres communs :

- La précision de la recherche : la distance maximale à l'objectif pour considérer le point comme atteint.
- Le temps de dérive du ballon : la durée entre deux changements de niveau de pression possibles (appelé $t_{\text{dérive}}$ par la suite).
- La durée d'exploration : la limite temporelle d'exploration à partir du temps de départ.

Étant donné les actions possibles pour un ballon, l'ensemble des chemins possibles est donc **un arbre dont la racine est le point de départ et dont chaque noeud a 17 fils** correspondant aux 17 niveaux de pression des données de la NOAA où celui-ci peut dériver. Les éléments de hauteur m dans l'arbre sont les points accessibles après m périodes de dérive, avec éventuellement un changement d'altitude entre chaque dérive. En notant n le nombre de dérives du ballon (à savoir la durée d'exploration divisée par le temps de dérive), il y a donc **17ⁿ trajectoires possibles** pour notre problème. L'exploration exhaustive de ces trajectoires est évidemment impossible, ce qui implique que nos algorithmes devront choisir judicieusement les trajectoires à explorer. Les algorithmes qui suivent présentent les différentes stratégies que nous avons développées pour sélectionner les points à explorer. Leur structure générale est toujours la suivante :

Algorithm 1 Recherche d'un chemin du départ à la destination

```

1: liste_points  $\leftarrow$  [départ]                                      $\triangleright$  Initialisation avec le point de départ
2: for  $i \in \{1, \dots, n\}$  do
3:   liste_points_atteints  $\leftarrow$  []                                 $\triangleright$  Initialise la liste des points atteints
4:   for all point  $\in$  liste_points do
5:     a_rencontre_destination, new_point  $\leftarrow$  parcours_interpolate(destination, point,  $t_{\text{dérive}}$ )
6:     if a_rencontre_destination then
7:       return true, chemin           $\triangleright$  Renvoie vrai et le chemin si la destination est atteinte
8:     end if
9:     liste_points_atteints.append(new_point)     $\triangleright$  Ajoute le nouveau point aux points atteints
10:   end for
11:   liste_points  $\leftarrow$  selection(liste_points_atteints)  $\triangleright$  Sélection des points pour l'itération suivante
12: end for
13: return false, []
  
```

Enfin, nous avons soumis chacun de ces algorithmes à **une série de tests**, certains communs et d'autres spécifiques afin d'estimer leurs performances. Le test commun consiste à évaluer les performances des algorithmes sur des trajets entre des villes françaises, puis européennes. Ces tests seront effectués avec les paramètres suivants :

Précision demandée	10 km
Temps de dérive $t_{\text{dérive}}$	6 heures
Durée d'exploration	120 heures

3.1 ALGORITHME "NAÏF" : LIMITÉ D'ÉLOIGNEMENT

3.1.1 • FONCTIONNEMENT

La première idée que nous avons développée a été d'instaurer une **limite d'éloignement**. Nous avons décidé de ne plus considérer les points s'éloignant trop de l'objectif. Dans ce cas, la fonction de sélection prend la forme suivante :

Algorithm 2 Sélection des points avec une limite d'éloignement

```

1: function SELECTIONELOIGNEMENT(liste_points, destination)
2:   res  $\leftarrow$  []
3:   for all point  $\in$  liste_points do
4:     if distance(point, destination)  $<$  limite_eloignement then
5:       res.append(point)
6:     end if
7:   end for
8:   limite_eloignement  $\leftarrow$  limite_eloignement  $\times$  facteur_retrecissement
9:   return res
10: end function

```

La limite d'éloignement est réduite d'un facteur constant à chaque appel de la fonction. Celui-ci est choisi de façon à ce que la limite d'éloignement finale vaille un quart de la distance entre le départ et la destination. Ces deux paramètres sont stockés en dehors de la fonction de sélection.

3.1.2 • RÉSULTATS

On obtient naturellement une complexité dans le pire des cas en 17^n avec n le nombre d'itérations de la boucle d'exploration. Un tel résultat est loin d'être satisfaisant mais nous permet d'obtenir nos premières trajectoires. Dans la majorité des cas, nos ordinateurs ne nous permettent pas plus de 6 itérations de la boucle principale, ainsi nous pouvons optimiser notre algorithme en ajustant le paramètre $t_{\text{dérive}}$, ainsi que la **limite d'éloignement**.

À titre d'exemple, voici un des premiers trajets que nous avons obtenus : un trajet de l'École polytechnique (située à Palaiseau) jusqu'à la montagne Sainte-Geneviève (qui accueillait l'Ecole polytechnique jusqu'en 1976).

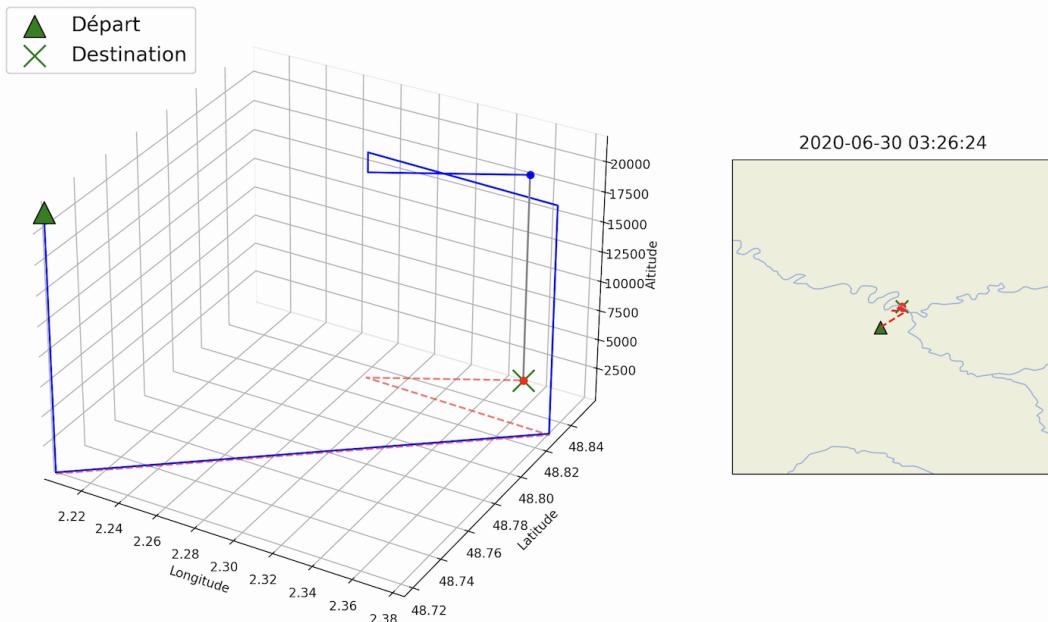


Figure 6: Trajectoire du campus de l'X (Palaiseau) à la montagne Sainte-Geneviève.

3.1.3 • BILAN

Comme cela a été dit, cette approche naturelle nous a permis d'obtenir nos premières trajectoires. Néanmoins, il est vite apparu que cette approche souffrait de plusieurs défauts importants :

- Il n'existe aucune garantie que l'algorithme termine en temps raisonnable. Il se peut qu'aucun point ne dépasse la limite d'éloignement et soit oublié.
- Les points "stagnants" qui n'évoluent peu car en proie à des vents de faible intensité pendant de longue durée ne sont pas éliminés.
- L'introduction de la limite d'éloignement ferme ainsi la voie à des trajectoires qui feraient potentiellement un détour avant de parvenir à l'objectif.

3.2 ALGORITHME "SÉLECTIF" : PARCOURS LINÉAIRE

3.2.1 • FONCTIONNEMENT :

Afin de palier le caractère exponentiel de notre premier algorithme, la méthode la plus naturelle était de **limiter le nombre de points explorés à chaque itération**. En effet, cela permet d'obtenir une complexité linéaire en le nombre d'itérations ($O(n)$), ce qui est primordial pour pouvoir étudier des trajets arbitrairement longs. Ce sera le cas dans les trois algorithmes qui suivront.

L'algorithme sélectif conserve à chaque itération **les N points les plus proches de la destination**. Plus précisément il sélectionne les N points les plus proches parmi les $17N$ points obtenus à partir des N précédents. Cette sélection se fait naturellement avec la méthode *quickselect*. Celle-ci prend en entrée une liste et un indice N et renvoie le N -ième élément de la liste dans l'ordre croissant. La fonction modifie la liste et place tous les éléments inférieurs au pivot à sa gauche et les éléments supérieurs à sa droite.

Algorithm 3 Sélection des points avec quickselect

```

1: function SELECTIONN(liste_points, destination,  $N$ )
2:   quickselect(liste_points,  $N$ , ordre = distance à la destination)
3:   return liste_points[0 :  $N$ ]           ▷ Récupère les  $N$  premiers éléments de la liste
4: end function
```

3.2.2 • RÉSULTATS

Cet algorithme offre une **demande en mémoire constante** et permet une **complexité linéaire** en le nombre d'itérations de la boucle de recherche.

Résultats obtenus sur les tests communs avec $N = 100$:

France	
Fréquence de chemins trouvés	57.48 %
Distance moyenne à la destination	149.224 km
Moyenne temporelle d'exécution	1.289 secondes
Europe	
Fréquence de chemins trouvés	48.5%
Distance moyenne à la destination	491.04 km
Moyenne temporelle d'exécution	10.5 secondes

3.2.3 • BILAN

L'écriture de cet algorithme linéaire nous a permis de chercher systématiquement des trajectoires avec une garantie de terminaison. Au-delà de ce critère de terminaison, cette stratégie linéaire reste assez performante en termes de chemins trouvés.

Enfin, bien que cette approche puisse ne pas garantir la meilleure solution possible, elle offre **une alternative efficace** pour des problèmes où l'exploration exhaustive est difficile à réaliser en raison de contraintes de temps ou de ressources.

3.3 ALGORITHME GLOUTON

3.3.1 • FONCTIONNEMENT

Après avoir conçu notre algorithme sélectif, nous avons pensé au cas particulier où $N=1$ de l'algorithme. Il fournit un algorithme glouton qui consiste à prendre à chaque étape la meilleure décision locale dans l'espoir d'atteindre un résultat global optimal. Dans notre contexte, cela signifie choisir à chaque instant **le point le plus proche de la destination**.

Algorithm 4 Sélection gloutonne

```

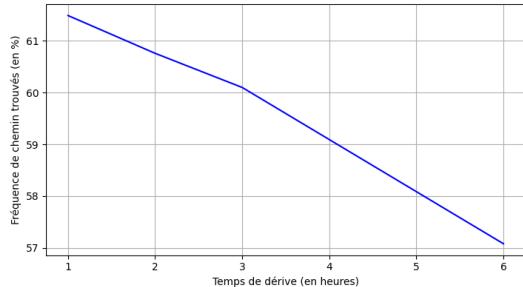
1: function SELECTIONGREEDY(liste_points, destination)
2:   return [min(liste_points, ordre = distance à la destination)]
3: end function
```

3.3.2 • RÉSULTATS

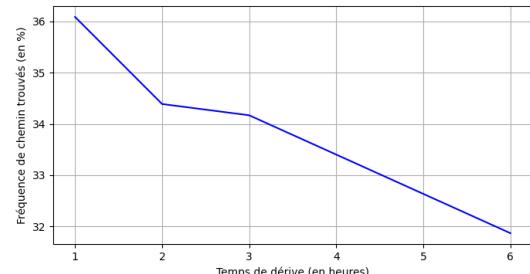
Résultats obtenus sur les tests communs :

France	
Fréquence de chemins trouvés	56.99 %
Distance moyenne à la destination	378,364 km
Moyenne temporelle d'exécution	0.112 secondes
Europe	
Fréquence de chemins trouvés	31.78%
Distance moyenne à la destination	730.043 km
Moyenne temporelle d'exécution	0.139 secondes

Dans l'algorithme glouton, le paramètre $t_{\text{dérive}}$ joue un rôle primordial en ce qu'il permet de déterminer la fréquence à laquelle le ballon peut se réorienter dans la bonne direction. Nous avons donc étudié l'influence de la variation de ce paramètre sur l'efficacité de l'algorithme.



(a) Résultats sur un ensemble de villes françaises.



(b) Résultats sur un ensemble de villes européennes.

Figure 7: Performance de l'algorithme Glouton en fonction de $t_{\text{dérive}}$

On remarque ainsi naturellement que plus le temps de dérive est court, au plus l'algorithme est performant. Cependant, l'écart de performance n'est pas conséquent entre $t_{\text{dérive}} = 6$ heures et $t_{\text{dérive}} = 1$ heures. Ceci n'est pas surprenant, étant donné que la résolution temporelle de nos données de vents est elle-même de 6 heures. De plus, dans le cadre de notre hypothèse de changement d'altitude en temps constant, nous devons nous assurer d'avoir un temps de dérive suffisamment grand par rapport au temps réel de changement d'altitude.

3.3.3 • BILAN

Cet algorithme glouton promet d'être à la fois **simple**, tant dans l'implémentation que dans la compréhension, mais aussi **efficace** car nous sommes assurés de progresser rapidement vers notre destination lorsque cela est possible. Cette approche est particulièrement avantageuse quand la notion de temps du trajet est importante, ce qui fait de cette approche une bonne solution au problème.

Cependant, cette approche présente également des défauts potentiels :

- Risque de sous-performance :** En se concentrant uniquement sur les choix locaux les plus favorables à chaque étape, l'approche gloutonne peut négliger des chemins qui, bien que moins directs, pourraient conduire à une solution globale meilleure.
- Biais dans les trajectoires trouvées :** L'ensemble des trajectoires obtenues par cet algorithme sont similaires dans la mesure où elles sont assez directes du départ à la destination. Cela a pu poser problème lorsque nous avons voulu constituer une base de chemins trouvés utilisables pour nos approches d'apprentissage supervisé.

3.4 ALGORITHME WIDE-SEARCH : MAXIMISATION DE L'ESPACE COUVERT

L'algorithme *wide-search* est l'algorithme classique (dans le sens où il n'utilise pas de méthodes d'apprentissage) **le plus précis** que nous avons développé. Son but est de réussir à trouver tous les types de trajectoires et pas nécessairement celles qui se rapprochent le plus possible à chaque étape. Néanmoins l'algorithme doit explorer le minimum de trajectoires pour assurer **une rapidité d'exécution satisfaisante**.

L'idée est d'assurer une couverture exhaustive de tous les trajets possibles en subdivisant l'espace en cases suffisamment petites par rapport à la précision demandée. Bien que le nombre de trajectoires existantes soit exponentiel, la surface de la Terre étant finie, certaines trajectoires seront inévitablement proches les unes des autres. Le but de cette subdivision est de réduire le nombre de trajectoires explorées en n'en gardant qu'une dans un rayon donné.

En pratique on ne sélectionne des points uniquement s'ils sont à une distance suffisamment grande des autres points choisis. Cette distance évolue comme une fonction croissante de la distance du point à la destination, typiquement la fonction racine dans notre étude³. Cette condition permet également de majorer le nombre de points qui sont explorés au sein d'une boucle par environ 10000 (une preuve est développée en annexe A.), ce qui est de l'ordre de grandeur souhaité pour obtenir un temps de calcul de l'ordre de la minute.

Algorithm 5 Sélection conditionnelle de points

```

1: function SELECTIONWIDE(liste_points, destination, precision)
2:   res  $\leftarrow \emptyset$ 
3:   for point  $\in$  liste_points do
4:     ajouter  $\leftarrow$  True
5:     for point_selectionne  $\in$  res do
6:       if  $d(\text{point}, \text{point\_selectionne}) \leq \sqrt{\text{precision} \times d(\text{point}, \text{destination})}$  then
7:         ajouter  $\leftarrow$  False
8:         break
9:       end if
10:      end for
11:      if ajouter then
12:        res.append(point)
13:      end if
14:    end for
15:    return res
16: end function
```

³Cela permet d'avoir une croissance assez faible et de garder des cases de taille raisonnable sur l'ensemble de la surface de la Terre.

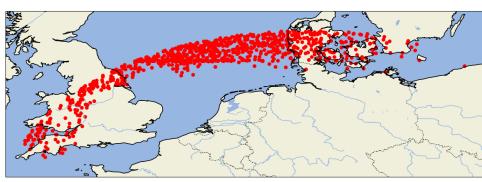
3.4.1 • RÉSULTATS

Résultats des tests communs en France et en Europe :

France	
Fréquence de chemins trouvés	83.0%
Distance moyenne à la destination	55.358 km
Moyenne temporelle d'exécution	85.91 secondes
Europe	
Fréquence de chemins trouvés	63.6%
Distance moyenne à la destination	319.778 km
Moyenne temporelle d'exécution	111.325 secondes

3.4.2 • BILAN

Les résultats prouvent que l'algorithme *wide-search* est **beaucoup plus performant** que les précédents. Son temps d'exécution est certes plus long mais il reste **linéaire en le nombre d'itérations**. Ainsi, l'algorithme reste tout aussi efficace sur des longs trajets.



(a) Cas de l'algorithme "sélectif" :



(b) Cas de l'algorithme *wide-search* :

Figure 8: Répartition des points en cours d'exploration lors de l'exécution de nos algorithmes.

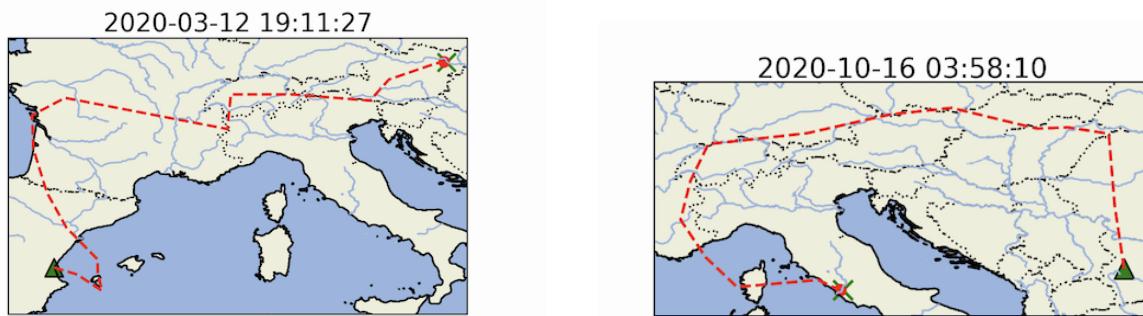
Contrairement aux algorithmes précédents, l'algorithme *wide-search* considère, à chaque itération, des points couvrant la surface la plus grande possible. Ainsi l'algorithme **trouve des trajectoires faisant de grands détours**, ce que ne pouvaient pas faire les algorithmes précédents. On peut visualiser les points en cours d'exploration grâce à une animation qui les affiche sur la carte. On peut donc suivre l'évolution du "nuage" de points et comprendre son comportement en fonction de la tendance générale des vents. Dans le cas où l'algorithme ne trouve pas de chemin, on peut toujours le raffiner en rajoutant une constante strictement inférieure à 1 dans la fonction racine. L'animation permet de visualiser l'effet de ce raffinement et d'identifier s'il est pertinent dans ce cas précis.

3.5 SYNTHÈSE SUR LA PERFORMANCE DES ALGORITHMES CLASSIQUES

Nos algorithmes, conçus pour optimiser la navigation des ballons stratosphériques, ont démontré une efficacité notable dans la prévision de trajectoires précises. Grâce à l'hypothèse de connaissance absolue des vents, nous sommes capables de déterminer avec exactitude les itinéraires les plus adéquats pour les ballons.

Toutefois, nos systèmes ne parviennent pas systématiquement à trouver une trajectoire viable pour chaque situation. Cette limitation soulève une question importante : l'absence de chemin identifié est-elle due à une réelle impossibilité de naviguer entre les points donnés à un moment précis, ou bien à une insuffisance de notre algorithme qui n'exploré pas toutes les options potentielles ? Cette distinction est cruciale pour l'amélioration continue de nos modèles et la compréhension des limites actuelles de nos algorithmes.

A titre d'exemple, nous avons recensé un certain nombre d'ensembles départ, arrivée, $date_{initial}$ pour lesquels l'algorithme glouton (et éventuellement l'algorithme selectif) ne trouve pas de chemin tandis que l'algorithme *wide-search* en trouve. Pour ces cas, comme nous l'avons expliqué dans la section 3.4.3, l'étude des vents appuie les limites des différents algorithmes.



(a) Trajectoire trouvée par l'algorithme sélectif et non par l'algorithme "Glouton".

(b) Trajet trouvé par l'algorithme *Wide-search* et pas par l'algorithme sélectif.

Figure 9: Différences de fiabilité entre nos algorithmes.

La question soulevée par les limitations de nos algorithmes actuels ouvre la voie à l'exploration de solutions via le *machine learning*. En intégrant le *machine learning*, nous pourrions non seulement améliorer la capacité de nos modèles à détecter et naviguer à travers des chemins complexes mais aussi mieux identifier si une situation donnée est réellement infranchissable ou simplement mal appréhendée par nos méthodes actuelles. Cette transition est essentielle pour transcender les contraintes actuelles et optimiser continuellement nos algorithmes de navigation.

4

APPROCHE PAR MACHINE LEARNING

En parallèle du développement de nos algorithmes traditionnels, il nous a semblé dès le départ indispensable de mettre en place d'autres méthodes basées sur le *Machine Learning*. En effet, dans l'optique d'une application concrète de nos recherches, il serait crucial de disposer de **méthodes capables de s'adapter à des contraintes plus importantes**. Le *Machine Learning* permet de traiter les problèmes déjà étudiés par les algorithmes classiques mais surtout ouvre la voie à la résolution du problème avec des hypothèses moins fortes.

Les équipes de Google, au sein du projet Loon, ont réussi à traiter le problème de **stabilisation un ballon au-dessus d'une zone géographique définie** en utilisant des réseaux de neurones simples entraînés par *Reinforcement Learning* [2]. Inspiré par leurs travaux, notre projet vise à améliorer et étendre ces techniques employées par les équipes de Google, pour **résoudre le problème plus complexe de navigation du ballon d'un point A à un point B**. Contrairement à la situation initiale qui se limitait à un contrôle localisé, notre défi implique de déterminer des itinéraires qui peuvent traverser de vastes étendues, souvent en incluant de **multiples changements d'altitude et donc de direction**. En effet, les résultats obtenus par les algorithmes précédents indiquent que les trajets les plus complexes, souvent omis par les approches plus simples, nécessitent parfois d'importants détours par rapport aux points de départ et d'arrivée. Ces itinéraires complexes se caractérisent par des parcours qui peuvent initialement s'éloigner de la destination finale avant de s'en rapprocher. Ceci suggère que **certains chemins encore plus complexes existent et n'ont pas pu être trouvés par les premiers algorithmes** en raison des limitations en termes d'exploration de l'espace. Notre objectif est de détecter et d'exploiter ces trajectoires à l'aide de modèles avancés basés sur des **réseaux de neurones**, espérant ainsi surmonter les limitations des méthodes précédentes et offrir des solutions de navigation plus efficaces et précises.

Pour ce faire, nous avons exploité les capacités des réseaux de neurones, en particulier **les réseaux de neurones convolutifs** que nous avons adapté au contexte de notre étude. Afin de les entraîner, l'apprentissage par renforcement nous a semblé particulièrement adaptée. Ce choix nous a incités à approfondir nos connaissances en *Reinforcement Learning* et en *Supervised Learning*, ainsi qu'à explorer les diverses branches constituant ces domaines. Notre objectif était de développer un outil spécifiquement adapté aux particularités de notre problème.

4.1 APPROCHE GÉNÉRALE

4.1.1 • OBJECTIF

Notre objectif est de développer un agent disposant, en entrée, des données relatives aux vents à un instant donné et à des instants futurs, ainsi qu'à sa propre position et celle de la destination finale. Cet agent doit être capable de **fournir en sortie la meilleure action à entreprendre pour atteindre le point cible en un temps fini**. Nous adoptons les mêmes hypothèses que celles utilisées dans les algorithmes précédents : **le changement d'altitude peut être considéré comme instantané**. De plus, nous partons du principe que les informations relatives aux vents sont parfaitement connues et précises à tout instant.

4.1.2 • LES DONNÉES

Comme mentionné précédemment, restreindre les données fournies au modèle à une zone spécifique peut limiter notre capacité à détecter certains des trajets les plus atypiques, surtout dans des scénarios défavorables. C'est pourquoi il est essentiel **d'inclure l'intégralité des données de vent à l'échelle mondiale** pour notre recherche de trajets. Pour ce faire, nous utilisons, comme dans le cadre des premières méthodes, les données fournies par la **NOAA**, qui se caractérisent par une structure à quatre dimensions : longitude, latitude, altitude, temps.

Bien qu'il soit possible d'augmenter significativement la taille de nos données en augmentant la précision sur les vents, via interpolation ou à partir de bases de données plus précises, il n'est pas évident que cette démarche améliore de façon significative nos résultats. En effet, si celle-ci repose sur l'interpolation, aucune information supplémentaire n'est en réalité fournie au modèle. De plus, les observations montrent que les variations de vent entre les points voisins de la grille ne sont généralement pas significatives.

Cette grande quantité de données pose un défi significatif en termes de modélisation. En effet, la haute dimensionnalité exige un nombre conséquent de paramètres à estimer et à optimiser lors de l'entraînement des modèles de réseaux de neurones. Cependant, si l'on ne dispose pas de suffisamment d'informations, l'agent ne sera pas en mesure d'évoluer convenablement dans son environnement. L'enjeu est donc de faire **un compromis en choisissant la bonne quantité de données à fournir au modèle**, ainsi que leur format, et de traiter ces données de manière judicieuse pour les condenser et simplifier leur gestion.

Une première réduction découle de l'observation suivante : le modèle peut être mis à jour aussi souvent que l'on souhaite, ce qui implique que seule la direction du déplacement importe. En conséquence, chaque couple de vitesse associé au vent en un point **est transformé en un angle, puis normalisé** pour être représenté par une valeur réelle $\theta_{\text{long},\text{lat},\text{alt}}$ dans l'intervalle [-1, 1]. La normalisation de nos données sera un moyen de faciliter leur traitement par nos modèles. Ainsi, nous **réduisons la taille des données** en ne conservant qu'une seule valeur par point.

4.2 APPRENTISSAGE SUPERVISÉ

Pour faciliter l'entraînement par apprentissage par renforcement, nous avons d'abord **pré-entraîné nos réseaux de neurones sur des trajectoires déjà connues**, établies grâce aux algorithmes traditionnels développés précédemment. Cette étape de pré-entraînement a permis à notre réseau d'adopter dès le départ un comportement approprié, fournissant une fondation solide pour des améliorations progressives. Notre objectif était de découvrir de nouveaux chemins potentiellement plus efficaces et de surpasser les performances des approches antérieures. En outre, cette phase de pré-entraînement s'est révélée **essentielle pour le développement et l'ajustement des architectures de nos réseaux de neurones**. Elle nous a permis de confirmer que les réseaux pouvaient apprendre et reproduire les comportements issus des méthodes classiques. En ajustant progressivement les hyperparamètres, les dimensions et les structures des réseaux, nous avons amélioré leurs performances jusqu'à atteindre des résultats jugés satisfaisants, que l'on tentera ensuite d'améliorer.

4.2.1 • RÉSEAUX DE NEURONES CONVOLUTIFS

Face à la grande taille de nos données, l'utilisation de réseaux de neurones linéaires, tels que ceux employés par les équipes de Loon [2], n'est pas adaptée. Une méthode **plus appropriée pour notre contexte est l'utilisation de Réseaux de Neurones Convolutifs** (CNN : *Convolutional Neural Networks*). Ces modèles, largement utilisés dans les tâches de reconnaissance d'image, ont conduit à des avancées majeures dans ce domaine. Nous avons adapté ces méthodes à la géométrie spécifique de notre environnement pour développer des réseaux capables d'identifier des chemins efficacement.

Les CNN reposent sur trois composantes principales qui sont cruciales pour leur efficacité et fonctionnalité :

Couches de convolution : Ces couches exploitent des filtres qui glissent sur l'ensemble de l'entrée pour générer ce que l'on nomme des (*feature maps*). Dans notre contexte, l'objectif est que ces filtres réussissent à **identifier des schémas** dans les données de vent qui peuvent être utilisés pour déterminer des trajets navigables. Les paramètres de chaque filtre, c'est-à-dire les coefficients, sont appris durant le processus d'entraînement.

Les filtres sont caractérisés principalement par leur taille et leur pas de glissement (*stride*). La taille du filtre détermine les dimensions du filtre appliqué sur chaque canal d'entrée, affectant directement le nombre de canaux en sortie en fonction du nombre de filtres employés. La convolution peut être exprimée par :

$$\text{Output}[i, j, l] = \sum_k \sum_{m,n} \text{Input}[i + m, j + n, k] \times \text{Kernel}[m, n, k, l]$$

où i, j sont les indices de l'emplacement dans l'image d'entrée, m, n sont les indices des coefficients du filtre, k l'indice des canaux en entrée et l l'indice du canal de sortie.

Dans notre projet, nous utilisons une **variété de dimensions de filtres** pour différentes applications:

- Filtres 1D à pas unitaire : Utilisés pour transformer les données d'entrée dans des espaces de dimensions plus grandes ou plus petites en termes de canaux.
- Filtres 2D à pas variable : Employés à la fois pour extraire des caractéristiques de l'image et pour réduire ses dimensions spatiales, par exemple, des filtres de taille 2×2 avec un pas de 2, 2 permettent une réduction dimensionnelle par sous-échantillonnage.
- Filtres 3D : Utilisés dans certains modèles pour traiter directement nos données en trois dimensions.

Couches de pooling (*MaxPooling* étant le plus commun) : Ces couches réduisent la dimensionnalité spatiale des cartes de caractéristiques tout en conservant les informations les plus importantes. Le *pooling* aide à rendre le réseau plus robuste aux petites variations et déformations dans l'image d'entrée en conservant par exemple le maximum parmi un bloc de pixels (*textit{MaxPooling}*) dans la carte de caractéristiques, ou alors la moyenne des valeurs (*AveragePooling*). Les caractéristiques de ces couches sont identiques à celles des filtres, et sont choisies en fonction des cas.

Fonctions d'activation : Typiquement, les fonctions d'activation comme ReLU (*Rectified Linear Unit*) sont utilisées après chaque couche de convolution. ReLU ajoute de la non-linéarité au modèle, permettant au réseau de capturer des relations complexes dans les données. Cette fonction présentant pour $x < 0$ une dérivée nulle, il est possible de faire face à des difficultés lors de l'entraînement par descente de gradient, c'est pourquoi nous utiliserons une variation à dérivée non nulle, le LeakyReLU, comme suggéré dans "*Empirical Evaluation of Rectified Activations in Convolutional Network*" [3].

Architecture globale : Un réseau de neurones convolutifs (CNN) classique se compose généralement d'une séquence de couches, alternant entre opérations de convolution pour identifier des caractéristiques, de fonctions d'activations et *max pooling* pour réduire la dimension. On obtient finalement un vecteur unidimensionnel, qui est ensuite fourni à un réseau de plusieurs couches entièrement connectées.

Logique derrière les CNN : Le but des CNN est d'identifier des *features* ou caractéristiques à différents niveaux de complexité et d'abstraction. Le réseau commence par apprendre de petits motifs locaux et les assemble progressivement en motifs plus grands et complexes grâce aux couches successives, tout en diminuant la résolution des données (*downsampling*) pour concentrer l'analyse sur les informations plus globales.

En intégrant ces éléments, les CNN démontrent une capacité remarquable à traiter des images de grande taille et de haute complexité. Nous prévoyons d'exploiter cette capacité dans le cadre de l'**identification des chemins à partir de nos données de vents**. Cette application des CNN sort des usages conventionnels typiquement associés à ces réseaux, car elle s'applique à l'analyse de données différentes de celle généralement traitées, méthode encore peu exploré avec cette technologie.

4.2.2 • LES OUTILS UTILISÉS DANS NOS MODÈLES

Normalisation par lots

Pour améliorer la **stabilité et l'efficacité** de notre apprentissage, nous avons eu recours à la normalisation par lots, plus communément appelée *BatchNorm*, introduite dans l'article "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" [4]. Cette méthode consiste à normaliser les activations en sortie de chaque couche pour obtenir une moyenne nulle et une variance unitaire. À chaque itération durant l'entraînement, on fournit un lot d'entrées (un *batch*) au modèle. Chaque couche de *BatchNorm* calcule ainsi la moyenne et la variance des activations de ce lot, puis normalise ces dernières selon la formule :

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

où x_i est l'activation avant la normalisation, μ_B est la moyenne calculée sur le *batch*, σ_B^2 est la variance calculée sur le *batch*, et ϵ est un petit nombre ajouté pour éviter la division par zéro. Les activations normalisées sont ensuite translatées et mises à l'échelle à l'aide de paramètres γ et β , qui sont appris durant l'entraînement, pour ajuster respectivement la variance et la moyenne des activations normalisées.

Cette méthode aide à **limiter les problèmes de disparition ou d'explosion des gradients**, permettant ainsi l'utilisation de taux d'apprentissage plus élevés, ce qui **accélère la vitesse d'entraînement**.

Dropout

Pour **limiter le sur-entraînement**, ce qui est d'autant plus important dans le cadre d'un pré-entraînement où il est essentiel que le modèle puisse s'adapter à de nouvelles données, nous avons recours à la méthode du *dropout* introduite dans "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" [5]. Cette approche consiste à désactiver aléatoirement certains neurones dans notre réseau pendant l'entraînement. Pour ce faire, nous sélectionnons des neurones à la sortie de chaque couche et les mettons à zéro avec une probabilité p (on choisit $p = 0.1$ dans nos modèles). Cela revient à "éteindre" le neurone, qui ne jouera donc aucun rôle dans cette itération de la rétro-propagation. L'intérêt principal de cette technique est de **rendre le réseau plus robuste** en empêchant les neurones de devenir trop dépendants les uns des autres, ce qui améliore la capacité de généralisation du modèle. Nous avons décidé d'appliquer un taux de *dropout* après chaque fonction d'activation dans tous nos entraînements.

Réseaux résiduels

Intuitivement, il est logique de penser que l'augmentation de la taille d'un réseau, et notamment de sa profondeur, favorise l'émergence de comportements plus complexes et, à terme, de meilleurs résultats. Cependant, le phénomène bien connu des "gradients qui s'évanouissent ou explosent" (*vanishing/exploding gradients*) pose un problème majeur. Ce phénomène désigne la tendance des gradients à tendre vers zéro ou l'infini lors de la rétropropagation à travers les couches profondes du réseau, rendant ainsi l'entraînement de ces réseaux profonds particulièrement difficile. Nous avons rapidement observé ce problème dans notre projet, constatant que **les modèles les plus profonds étaient paradoxalement moins performants que prévu**.

Pour pallier ce problème, de nouvelles architectures de réseaux ont été introduites dans l'article *"Deep Residual Learning for Image Recognition"* [6] : les réseaux résiduels, ou *ResNets*. Le principe de ces réseaux repose sur **l'apprentissage d'un résidu plutôt que de la sortie complète** grâce à des connexions résiduelles. La composante clé de ces réseaux est le Bloc Résiduel. Ce bloc est constitué de couches convolutives classiques, mais se distingue par **l'ajout de connexions directes entre les blocs résiduels**, également appelées connexions résiduelles, qui consistent à additionner l'entrée de ce bloc à sa sortie. Formellement, si l'on cherche à apprendre une fonction $H(x)$ où x est l'entrée d'un bloc, les couches du bloc apprennent une fonction $F(x)$, de sorte que la sortie du bloc est $x + F(x) = H(x)$. F , qui est donc le résidu que l'on souhaite obtenir, devrait être a priori plus simple à apprendre. On applique à la sortie des blocs, comme à chaque fois, une fonction d'activation, qui ici est le ReLU.

En pratique, nous utilisons des blocs composés de deux couches de convolution. On distingue deux types principaux de blocs résiduels :

- Les blocs résiduels classiques, qui ne modifient pas la taille des données.
- Les blocs résiduels avec projection, qui réduisent la taille des données tout en augmentant notamment le nombre de canaux. Il est alors nécessaire de réduire la taille de l'entrée, via une projection obtenue en appliquant une *average pooling* 2D pour réduire la taille, puis une convolution 1D pour augmenter le nombre de canaux.

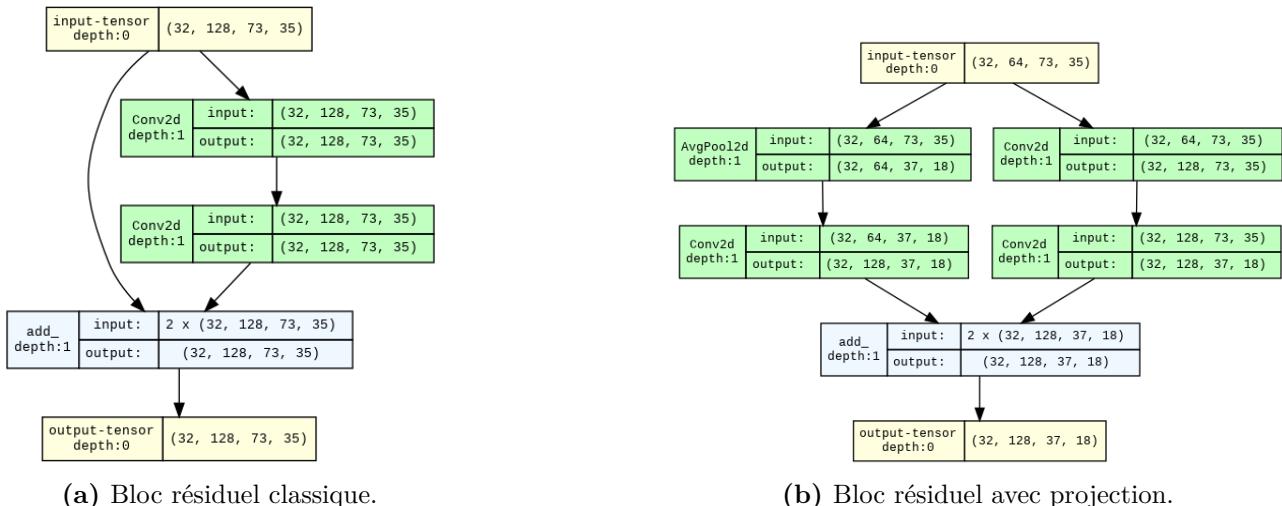


Figure 10: Blocs élémentaires pour les ResNet

Ces blocs résiduels nous ont ainsi permis d'entrainer des réseaux de neurones plus profonds, et d'améliorer les performances globales.

4.2.3 • LES MODÈLES

Convolution 3D sur vents bruts : 3D CNN

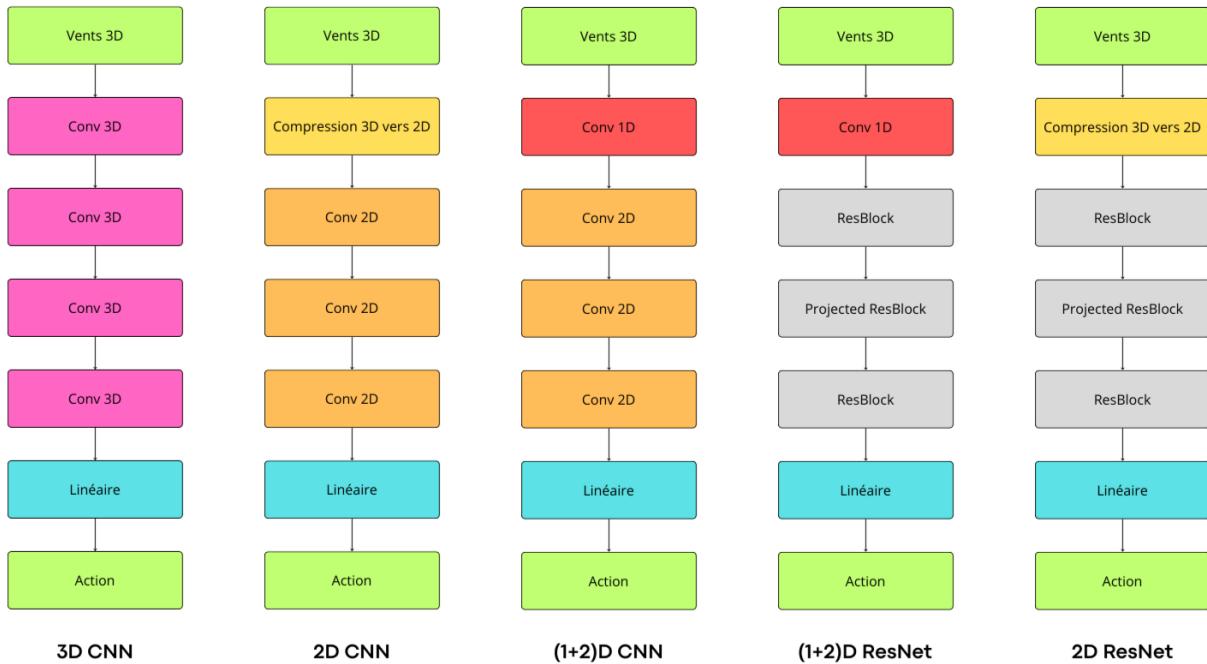


Figure 11: Schémas des différents types de modèles explorés

La première classe de modèles utilise **des données de vents bruts, sans traitement préalable**. Ces modèles reçoivent une grille en trois dimensions, comportant deux canaux : un canal pour les vents, encodant leur angle, et un autre canal pour la position du ballon ainsi que celle de l'objectif, ces derniers étant représentés respectivement par un 1 et un -1 dans la grille, à l'emplacement correspondant à leur position dans l'espace. La seule modification apportée à ces données une fois constituées était une augmentation selon la latitude pour prendre en compte la géométrie sphérique de la Terre. Cette opération consistait à étendre chaque bord de la grille de deux unités en utilisant les données de l'autre bord, afin de simuler la continuité du globe terrestre. Cette méthode reprend les idées introduites dans l'article "*Optimal path finding with space- and time-variant metric weights via multi-layer CNN*" [7] où la position et les obstacles sont encodés dans des canaux différents.

L'architecture du premier modèle adopté est relativement simple et se compose d'une **série de couches que l'on retrouve classiquement dans un CNN**. Cela inclut des couches de convolution, des couches de *pooling*, suivies d'un réseau entièrement connecté. La principale distinction avec les modèles classiques réside dans **l'utilisation de filtres tridimensionnels (3D)**, ce qui augmente significativement la complexité du modèle. Ces filtres 3D permettent une **meilleure intégration et analyse des données spatiales**. En sortie, ce modèle génère un vecteur à trois coefficients, chacun correspondant à une des trois actions possibles : monter d'un niveau d'altitude, descendre d'un niveau d'altitude, ou se laisser porter par le vent. En ajoutant une dernière fonction *SoftMax*, les trois coefficients obtenus correspondent à des probabilités associées à chaque action, qui peuvent être interprétées comme la probabilité que cette action soit la bonne.

Conformément aux recommandations de la littérature, les dimensions sont divisées par 2 à chaque augmentation du nombre de canaux, **dans le but de concentrer la complexité à mesure que**

I'on progresse dans le réseau. Ainsi, le modèle comprend une série de 7 couches conventionnelles suivies de 3 couches totalement connectées avant de produire un vecteur de taille 3 en sortie. À chaque sortie de couche, on applique un *BatchNorm*, ainsi qu'un dropout comme présenté dans les parties précédentes.

Convolution 2D sur vents traités 2DCNN

Le principal défaut du modèle précédent réside dans sa manière de considérer les différents niveaux d'altitude comme des couches distinctes disposées dans un ordre spécifique. Cette configuration rend **difficile l'établissement de liens entre les couches extrêmes**, la plus haute et la plus basse, par le modèle. Cependant, en vertu des hypothèses sur notre environnement, chaque couche remplit le même rôle, l'ordre des couches n'a aucune importance, et toutes sont mutuellement accessibles. En d'autres termes, **tout se passe comme si le ballon était présent sur toutes les couches simultanément**, ou que toutes les couches étaient fusionnées.

Pour intégrer cette caractéristique, et réduire la dimensionnalité, nous traitons les vents avant de les fournir au modèle afin d'en extraire manuellement des informations qui pourraient être utiles lors de l'apprentissage. Pour cela, nous avons **discrétisé les angles des vents** fournis en entrée du réseau de neurones. Ce processus permet de reformater les données de vent en une structure bidimensionnelle avec d canaux, chacun correspondant à l'une des d directions possibles. Dans cette configuration, chaque position sur un canal i de l'intervalle $[0, d-1]$ contient un 1 si la direction correspondante est présente dans la colonne de vent associée, et un 0 sinon. À cela s'ajoute un canal supplémentaire pour la position du ballon et celle de l'objectif, comme dans les modèles précédents. On élimine ainsi la distinction entre les couches d'altitude. Ce **traitement des données permet de ramener à une géométrie bidimensionnelle**, facilitant ainsi l'analyse par le réseau.

Ce format s'apparente à la résolution d'un labyrinthe : pour $d = 4$, par exemple, chaque canal encode la présence ou non d'un obstacle dans sa direction. Cette analogie suggère que cette méthode pourrait aider le modèle à mieux comprendre et interagir avec son environnement. Par ailleurs, l'article "*One-Shot Multi-Path Planning Using Fully Convolutional Networks in a Comparison to Other Algorithms*" [8] montre qu'il est possible de résoudre de tels labyrinthes à l'aide de CNN, ce qui laisse penser que cette méthode a des chances de fonctionner.

Le réseau lui-même est constitué de couches convolutives 2D, suivies de couches de *maxpooling*, et se conclut par un réseau entièrement connecté. En sortie, le modèle génère un vecteur de d coefficients qui sont ensuite traités par la fonction *SoftMax*, transformant chaque coefficient en une probabilité que cette direction soit la plus appropriée. Finalement, une fonction **détermine le niveau d'altitude dans la colonne de vent qui correspond le mieux à la direction ayant la plus haute probabilité**. Ce processus revient à discrétiser les angles possibles pour limiter les sorties du modèle.

Ainsi, plus le nombre d est élevé, plus la sortie est précise, mais plus l'entraînement devient complexe. Nous avons opté pour $d = 8$ ce qui offre un **bon équilibre entre la précision des prédictions et la complexité de l'entraînement**, tout en assurant une performance optimale du système dans divers scénarios de navigation.

Convolution (1+2)D sur vents bruts : (1+2)DCNN

Ce modèle s'inscrit dans la lignée du précédent, et cherche à **réduire la dimension des données**.

Pour ce faire, une première modification a été apportée au modèle 3DCNN : la position du ballon et de l'objectif sont encodées sur l'ensemble d'une colonne de vent dans le canal correspondant, **éliminant ainsi la distinction entre les couches d'altitude**. Le modèle commence ensuite par une couche de convolution 1D le long de l'axe de l'altitude, compressant ainsi toutes les altitudes en une série de caractéristiques bidimensionnelles. Ces données sont ensuite traitées par une série de couches conventionnelles 2D et de *maxpooling*, suivies d'un réseau entièrement connecté. Cette démarche s'inspire notamment des techniques développées dans "A Closer Look at Spatiotemporal Convolutions for Action Recognition" [9] pour le traitement de vidéos, afin de prendre en compte une dimension supplémentaire des données, mais dont le rôle est différent des deux autres. En sortie, le modèle génère encore vecteur de d coefficients, où d représente le nombre de directions possibles. Contrairement au modèle précédent, la **compression des altitudes est apprise par le réseau**, et un plus grand nombre de caractéristiques peuvent être extraites, puisque l'on peut engendrer des données avec un nombre de canaux bien plus grand que d .

Convolution et Réseaux Résiduels : 2DResNet et (1+2)DResNet

Les deux précédents modèles pouvant être **limités par la profondeur du réseau**, en raison des phénomènes évoqués précédemment dans le cadre de réseaux profonds, nous avons **remplacé les couches conventionnelles par des couches résiduelles**, ce qui nous a notamment permis d'augmenter leur nombre et par conséquent la profondeur.

L'architecture repose sur les deux types de blocs résiduels introduits plus tôt. On alterne alors entre des ensemble de 2 blocs résiduels classiques, entrecoupés de blocs résiduels avec projection, et ce, jusqu'à obtenir un vecteur à fournir au réseau totalement connecté.

4.2.4 • MÉTHODE D'APPRENTISSAGE

Le vecteur obtenu en sortie de nos modèles est d'abord traité par une fonction *Softmax* afin de convertir les *logits* (valeurs brutes prédictes par le modèle) en une distribution de probabilités sur l'ensemble des actions possibles. La fonction *Softmax* transforme efficacement les logits en probabilités en utilisant l'expression suivante :

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

où :

- z_i représente le score ou la logit de la classe i avant l'application de *Softmax* .
- K est le nombre total de classes.

Le dénominateur, $\sum_{j=1}^K e^{z_j}$, est la somme des exponentielles des scores de toutes les classes, assurant que les sorties du *Softmax* sont normalisées pour former une distribution de probabilité valide (la somme des probabilités est égale à 1).

La performance de notre modèle est ensuite évaluée à l'aide d'une fonction de perte spécifique, la *cross-entropy loss*. Cette fonction est particulièrement adaptée pour les problèmes de classification où il est nécessaire de choisir parmi plusieurs actions possibles, dans notre cas d actions. La *cross-entropy loss* mesure la différence entre les distributions de probabilité prédites par le modèle et la distribution réelle des étiquettes. La formule générale pour la *cross-entropy loss* est la suivante :

$$L = - \sum_{i=1}^N \sum_{c=1}^C y_{ic} \log(p_{ic})$$

où N est le nombre total d'exemples dans le lot, C est le nombre d'actions possibles, y_{ic} est un indicateur binaire (0 ou 1), indiquant si l'action c est correcte pour l'observation i , p_{ic} est la probabilité prédite par le modèle.

Cette erreur est calculée lors d'évaluations sur des lots de données (*batches*), typiquement de taille 32 ou 64. Nous avons observé empiriquement que **plus le lot est grand, plus l'entraînement tend à être stable**, probablement parce que des lots plus grands fournissent une meilleure estimation du gradient moyen sur l'ensemble de données. Chaque parcours de l'ensemble d'entraînement est une époque, et nous entraînons nos modèles typiquement sur 10 époques.

Pour optimiser les paramètres de nos modèles, nous utilisons la **méthode de rétro-propagation** pour calculer les gradients de chaque paramètre par rapport à la perte, à l'aide de la différentiation automatique fournie par les librairies classiques (PyTorch dans notre cas), puis nous appliquons un **algorithme de descente de gradient** pour ajuster les paramètres. Nous avons choisi d'utiliser l'optimiseur *Adam*, qui est largement répandu dans la littérature et s'est révélé être le plus efficace parmi ceux que nous avons testés. *Adam* combine les avantages de deux autres extensions de la descente de gradient stochastique : l'adaptation du taux d'apprentissage pour chaque paramètre (*AdaGrad*) et la conservation d'un moment de gradient pour accélérer la convergence (*RMSProp*).

Pour ajuster le taux d'apprentissage, nous avons opté pour un taux variable à l'aide d'un *scheduler* de taux d'apprentissage, plus précisément le modèle de *decay step*. Ce *scheduler* réduit le taux d'apprentissage par un facteur fixe après un nombre prédéfini d'époques, ce qui permet d'affiner les paramètres du modèle en effectuant des ajustements plus petits à mesure que l'on se rapproche d'un minimum local lors de l'optimisation.

4.2.5 • IMPLÉMENTATIONS ET RÉSULTATS

Un vaste éventail d'architectures a dû être exploré pour chacun des types de modèles mentionnés précédemment, afin d'**identifier les paramètres optimaux parmi les nombreuses possibilités disponibles**. Dans cette section, nous exposerons les meilleurs résultats obtenus pour chaque modèle, avec les architectures présentées dans la partie précédente.

Les entraînements furent réalisés à partir de **données générées par nos algorithmes traditionnels**, initialement dans le contexte de l'identification de chemins en France, puis étendus à l'ensemble de l'Europe. Les trajets initiaux reliaient des points distants en moyenne de 600 km, tandis que les trajets européens étaient significativement plus longs, pouvant s'étendre sur plusieurs milliers

de kilomètres. Au total, près de 50 millions de quadruplets (*objectif, position, action, temps*) ont été rassemblés, répartis de manière aléatoire en deux ensembles dans une proportion de 90/10. **Le premier ensemble est utilisé pour l'entraînement du modèle, tandis que le second sert à l'évaluation.** Nous avons pris en compte à la fois des chemins simples, que l'algorithme "glouton" était capable de trouver, ainsi que des chemins complexes identifiés uniquement par l'algorithme *wide-search*. L'objectif était de **développer un modèle capable de surpasser les performances des algorithmes traditionnels, même dans des situations complexes.** Ces chemins pouvant être générés sans limites, nous avons augmenté et élargi les données d'entraînements en continu.

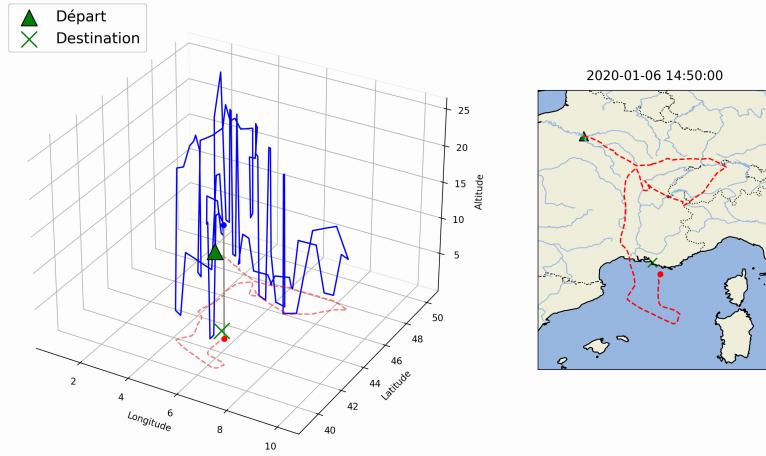


Figure 12: Chemin difficile liant Paris à Marseille trouvé par le modèle (2+1)DResNET

Les données d'entraînement étaient initialement générées en produisant un grand nombre de chemins entre deux points fixes. Cependant, nous avons observé qu'avec cette méthode, bien que le modèle présente des taux de précision extrêmement élevés ($> 95\%$), ce qui signifie qu'il apprend correctement les trajets précis, les résultats pour la découverte de nouveaux chemins sont extrêmement faibles ($< 10\%$). En effet, dès que le modèle se retrouve dans une position qui s'éloigne trop des positions contenues dans les chemins d'entraînement, il tend à diverger et à ne pas atteindre l'objectif. Nous avons donc opté pour **des chemins provenant d'un grand nombre de positions de départ vers une même destination.** Ainsi, le modèle est capable de se diriger vers un point donné quelle que soit sa position relative à celui-ci. On obtient alors des taux de précision légèrement inférieurs, mais **des taux d'identification de chemin considérablement améliorés.**

Nous avons fixé le taux d'entraînement de *Adam* à 10^{-4} , avec une réduction de moitié à chaque époque dans le cadre de la diminution progressive du taux d'apprentissage. Dans les cas les plus complexes, le taux d'apprentissage initial était de 10^{-5} . Nous avons utilisé des lots de données de taille 32 pour les neuf premières époques, puis des lots de taille 64 pour la dernière.

Les critères d'évaluation adoptés sont les suivants :

- **Précision des actions choisies :** évaluation du taux d'actions correctes prises par le modèle sur l'ensemble d'entraînement.

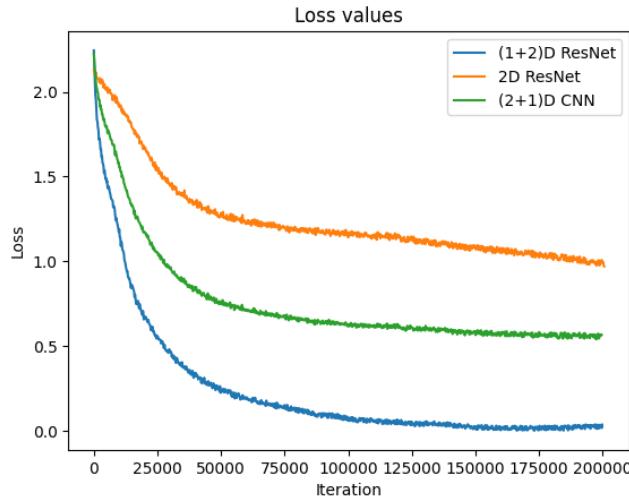


Figure 13: Erreurs calculées lors des entraînements

- **Taux d'identification des chemins pour des trajets fixés à des temps aléatoires :** on considère que le ballon atteint sa destination lorsque sa distance par rapport à l'objectif est inférieure à une distance seuil D , telle que les algorithmes traditionnels puissent se rapprocher autant que nécessaire de l'objectif à moindre coût, avec une quasi-certitude d'atteindre la destination aussi près que souhaité.

Algorithm 6 Navigation du ballon

```

1: while distance au point d'arrivée >  $D$  et  $t <$  limite_temps do
2:    $Direction \leftarrow$  Model(données de l'environnement)
3:    $Altitude \leftarrow$  Altitude à laquelle les vents correspondent le mieux à la direction souhaitée
4:    $t \leftarrow t + dt$ 
5:   Passer à cette altitude
6:   Dériver pendant un temps  $dt$ 
7: end while
  
```

Dans la pratique, la **distance D est fixée à 40 km**, ce qui correspond approximativement aux limites de précision de nos données : le modèle n'est pas capable de distinguer deux positions différentes si elles sont espacées d'une distance inférieure. De plus, dans de telles conditions, les algorithmes naïfs fournissent des performances satisfaisantes. Le temps de dérive dt varie en fonction de la distance à l'objectif, décroissant à mesure que l'on s'en rapproche, afin d'augmenter progressivement la précision.

Les modèles mentionnés précédemment **ne prenaient pas en compte les évolutions futures des vents**, une limitation significative pour la prédiction à long terme. Cependant, certains trajets peuvent durer plusieurs jours, et donc dépendre de données futures des vents. Pour remédier à cela, **nous avons exploré deux approches permettant d'intégrer les données sur une durée étendue**. Nous désignons par Δt le nombre de prédictions futures à considérer, espacées de 6 heures dans notre base de données. Ainsi, notre donnée d'entrée acquiert une dimension supplémentaire : nous disposons de Δt tableaux, chacun similaire au précédent mais avec des données de position nulles pour $t \geq 1$. Ces méthodes ont été appliquées uniquement aux trajets à l'échelle européenne, puisque

France				
	3DCNN	2DResNet	(1+2)DCNN	(1+2)ResNet
Fréquence de chemins trouvés	5%	20%	32%	49%
Taux de précision	41%	75%	89.0%	96%
Europe				
	3DCNN	2DResNet	(1+2)DCNN	(1+2)DResNet
Fréquence de chemins trouvés	2%	16%	19%	38%
Taux de précision	39%	69%	82%	91%

Table 1: *Résultats*

c'est sur ceux-ci que le manque d'information future pose le plus de problèmes compte tenu de leur longueur.

Première Approche : Concaténation

La première méthode **implique le traitement séparé de ces tableaux à travers un même réseau convolutif**. Après traitement par le réseau, les vecteurs de sortie de chaque tableau sont concaténés et ensuite passés à un réseau totalement connecté. C'est donc à la dernière partie du réseau de traiter la composante temporelle afin de prendre en compte les données futures. Cette stratégie augmente considérablement la taille de la couche entièrement connectée, ce qui augmente l'instabilité durant l'entraînement, et le rend plus long.

Seconde Approche : Convolution Temporelle

La seconde méthode s'inspire encore de l'article *A Closer Look at Spatiotemporal Convolutions for Action Recognition* [9] et consiste à appliquer des convolutions 1D le long de l'axe temporel, en complément des convolutions 2D appliquées de manière identique sur les Δt tableaux. **Cette approche permet de limiter considérablement l'augmentation de la dimensionnalité par rapport à la méthode précédente**. Il s'avère cependant préférable d'introduire les convolutions 1D uniquement dans les couches plus profondes du réseau pour optimiser l'intégration temporelle sans perturber les premières reconnaissances de motifs spatiaux.

Concaténation			
	2DResNet	(1+2)DCNN	(1+2)ResNet
Fréquence de chemins trouvés	13%	14%	34%
Taux de précision	60%	81%	88%
Convolution Temporelle			
	2DResNet	(1+2)DCNN	(1+2)DResNet
Fréquence de chemins trouvés	8%	15%	21%
Taux de précision	57%	75%	87%

Table 2: *Résultats*

4.2.6 • BILAN

Nous avons réussi à entraîner nos modèles en utilisant les résultats obtenus par nos algorithmes traditionnels, ce qui nous a permis d'atteindre des performances comparables à ceux-ci. Il est important de souligner que **notre modèle est particulièrement bien adapté pour une modélisation plus réaliste**, notamment dans les cas où les informations sur les vents sont partiellement connues. En effet, lorsqu'on dispose uniquement de distributions de probabilités des vents pour chaque point sur le globe, notre modèle peut se baser sur la moyenne de cette distribution pour fonctionner avec les mêmes paramètres d'entrée.

De plus, l'exploration heuristique joue un rôle crucial dans la gestion des incertitudes environnementales, **en permettant une navigation plus robuste face à une variance des vents qui peut être élevée**. Il serait également envisageable d'augmenter la précision de nos données d'entrée, ce qui nécessiterait l'utilisation de réseaux plus larges et plus profonds. Cependant, grâce à l'utilisation d'architectures avancées telles que les *ResNets*, nous pouvons anticiper que cette augmentation de la complexité des données n'empêcherait pas d'atteindre, voire de surpasser, les résultats actuels.

Cette approche, combinant des données améliorées et des techniques de modélisation sophistiquées, **offre une voie prometteuse pour améliorer la précision et la fiabilité de nos modèles dans des scénarios réels de navigation aérienne**.

4.3 APPRENTISSAGE PAR RENFORCEMENT

4.3.1 • RAPPEL SUR LE CADRE CONCEPTUEL DU REINFORCEMENT LEARNING

Dans un premier temps, dans un souci de clarté, rappelons brièvement quels sont les fondements théoriques du *Reinforcement Learning*, en les plaçant dans le cadre de notre étude.

Le *Reinforcement Learning* est **une méthode d'apprentissage** dans laquelle un **agent** - dans notre cadre, l'algorithme dirigeant le ballon - est plongé dans un **environnement** donné. Il apprend alors à prendre des décisions - pour un ballon, modifier son altitude - en fonction de l'état dans lequel il se trouve - ici, la position du ballon dans l'espace, ainsi que les informations sur les vents qui l'entourent. Cet apprentissage se fait de façon à **maximiser la récompense** reçue au cours du temps, qui s'interprète comme une mesure de l'accomplissement de la tâche qu'on lui a fixé : pour notre étude, réussir à effectuer un trajet.

Le **cycle crucial** intervenant dans le *Reinforcement Learning* est ainsi le suivant : l'agent est informé de l'état dans lequel il se trouve, il décide alors d'une action à effectuer, ce qui agit sur son environnement. Cette action va ainsi amener l'agent à accéder à un nouvel état, pour lequel il recevra une récompense.

Ainsi, l'objectif final du *Reinforcement Learning* va être de calculer une **politique** $\pi : S \rightarrow A$ qui associe à chaque état l'action optimale à effectuer lorsque l'on s'y trouve.

Séduisante sur le papier, cette méthode d'apprentissage machine est néanmoins source de nombreux défis dans son implémentation, qui nous amènent à introduire de nouveaux paramètres. Ainsi, dès la première page de leur article séminal sur l'utilisation du *Reinforcement Learning* pour apprendre à jouer à des jeux vidéos comme Atari, les équipes de DeepMind [10] éclarent : "RL algorithms [...] must be able to learn from a scalar reward signal that is frequently sparse [...] and delayed."

L'utilisation de ces termes de "*sparse*", c'est à dire clairsemé, et "*delayed*", retardé, illustre un phénomène important à l'œuvre dans les problèmes que l'on tente de résoudre avec du *Reinforcement Learning* : l'impact dans le temps d'une décision. En effet, certains choix en apparence anodins : modifier ou non son altitude à un instant donné alors qu'on est éloigné de la position cible, **peut parfois avoir un impact crucial sur le succès futur de l'agent**. Néanmoins, ceci a lieu sans que celui-ci ait une récompense instantanée pour une décision qui lui permettra plus tard d'arriver à destination.

Dès lors, résoudre cette contrainte nous amène à introduire un **facteur de réduction** $\gamma < 1$ dont le rôle sera de prendre ainsi en compte les récompenses obtenues plus tard dans le temps.

Ceci nous permet d'obtenir une métrique quantifiant le bon apprentissage de l'algorithme, **le gain G** que l'on définit de façon récurrente via la formule : $G_t = R_{t+1} + \gamma G_{t+1}$, où R_t désigne la récompense reçue après avoir effectué une action à l'instant t . Ainsi, **la politique optimale sera celle qui maximisera ce gain**.

4.3.2 • DESCRIPTION DE L'ENVIRONNEMENT ET DU SYSTÈME DE RÉCOMPENSE

Dans le cadre de nos travaux, il a été nécessaire de **construire un environnement de travail adapté au cadre théorique du RL décrit précédemment**. Pour cela, nous avons été amenés à utiliser l'interface de programmation d'application Gymnasium, développé par OpenAI, qui est l'une des plus communément utilisées en RL. Un état correspond ainsi à un vecteur dont les composantes sont celles mentionnées dans la section 4.1.2. À chaque état, il existe trois actions possibles : rester à la même altitude, l'augmenter ou la diminuer. Ce cadre de travail est identique à celui utilisé dans les quelques travaux modélisant déjà le déplacement des ballons atmosphériques [11].

Enfin, il nous a fallu choisir la fonction de récompense utilisée. Le cadre de notre problème étant **différent de celui de tous les travaux déjà effectués**, dans la mesure où notre objectif était de permettre à un ballon de se déplacer d'un point A à un point B, et non plus de se maintenir au-dessus d'une zone donnée, nous avons dû décider de notre propre système de récompense. Contrairement à ce qui est fait dans la littérature [11] pour maintenir un point au-dessus d'une zone, nous n'avons pas utilisé de pénalisation exponentielle de l'éloignement, celle-ci n'aurait pas été assez "lisse" pour permettre à notre agent de trouver une stratégie optimale. Notre méthode a plutôt été la suivante : dès lors que le ballon se trouvait suffisamment proche de sa destination (40 kilomètres), on lui attribuait une récompense de 1. Sinon, la récompense obtenue faisait intervenir deux sous-récompenses en distance et en temps de façon à inciter l'agent à se rapprocher au plus possible de son objectif en un minimum de temps.

$$r(d, t) = \begin{cases} 1 & \text{si } d < 40 \\ \max(0, 1 - \beta_d * (d - 40)) * \max(0, 1 - \beta_t * t) & \text{sinon} \end{cases}$$

Les paramètres β_t et β_d étant par ailleurs réglables afin de trouver le meilleur compromis entre exploitation et exploration.

4.3.3 • UTILISATION DU *DEEP Q-LEARNING*

Une fois notre environnement modélisé, il nous a fallu réfléchir aux méthodes de RL à implémenter afin de résoudre notre problème. Cette phase a été cruciale dans la mesure où ce domaine comprend de nombreux algorithmes plus ou moins adaptés à un problème donné. Dans un premier temps, de l'étude de la littérature générale sur le RL [12] nous est apparu qu'une méthode en particulier sortait du lot, de part sa facilité d'utilisation, son efficacité, et son adéquation avec notre problème : le *Deep Q-Learning* (DQL). En effet, dans le contexte qui est le nôtre, c'est à dire avec un espace d'actions discrétilisé mais un environnement de grande taille.

Apparu en 2014 et développé par les équipes de Google DeepMind [10], le DQL est l'une des formes d'apprentissage par renforcement profond la plus précoce. Elle s'inspire du Q-Learning, méthode développée bien plus tôt dans l'histoire par Christopher Watkins. [13] Dans cette méthode, une politique π étant fixée, on cherche à calculer pour chaque couple (Etat = s , Action = a), la valeur de l'espérance conditionnelle du gain futur après avoir effectué l'action a dans l'état s . D'un point de vue formel :

$$Q(s, a) = E_{\pi}(G_t | S_t = s, A_t = a).$$

L'apprentissage de la valeur de $Q(s, a)$ se fait ensuite par exploration de l'environnement. La politique initiale π cherche un compromis entre exploitation et exploration, et est souvent ϵ -greedy. Une fois les valeurs adéquates de la matrice Q apprises, la politique utilisée sera de choisir, à chaque état s , l'action a_{max} qui maximise la valeur de Q .

Bien qu'intuitive, cette méthode n'est pas utilisable en pratique lorsque la taille de la matrice Q est trop importante, comme c'est le cas dans le contexte de notre étude. Néanmoins, le DQL permet de passer outre cette difficulté, via l'utilisation de l'apprentissage profond et des réseaux de neurones.

En effet, l'idée principale fondant le *Q-Learning* est d'approximer la Q-fonction : on s'intéresse à $Q_{\theta}(s, a)$ où θ est un paramètre lié à un réseau de neurones.

Dans le cadre de notre projet, nous avons voulu réutiliser la structure du réseau de neurones convolutif déjà développé dans le cadre du *Supervised Learning*, ce choix était cohérent à la fois avec la littérature et avec la volonté d'exploiter une partie de notre travail déjà effectué. Il a néanmoins fallu modifier légèrement les dernières couches de celui-ci dans la mesure où pour le DQL le réseau de neurones a pour fonction de fournir en sortie les Q-valeurs estimées de chaque action et non pas la probabilité de celle-ci.

De nombreuses versions du DQL ont été proposées dans la littérature, nous avons choisi de travailler avec le même type d'algorithme que celui proposé par [14], dont le fonctionnement est recapitulé ci-dessous. Ce choix provient principalement du fait qu'il a été abondamment commenté et étudié par la communauté scientifique, et donc aisément à implémenter, ce qui n'est pas le cas pour les algorithmes les plus proches de l'état de l'art.

Algorithm 7 Q-Learning

```

1: Initialiser la mémoire D qui va stocker les transitions d'états
2: Initialiser la fonction Q avec des poids aléatoires  $\theta$ 
3: Initialiser la fonction objectif  $Q_*$  avec des poids  $\theta_* = \theta$ 
4: for episode = 1, M do
5:   Initialiser l'état de départ  $S_0$ 
6:   for t = 1, T do
7:     Sélectionner une action aléatoire  $a_t$  avec probabilité  $\epsilon$ 
8:     Sinon, choisir  $a_t = argmax_a Q(s, a, \theta)$ 
9:     Executer l'action  $a_t$  et obtenir  $r_t$  et  $s_{t+1}$ 
10:    Stocker  $x_t = (a_t, r_t, s_{t+1})$  dans D
11:    Choisir une transition aléatoire  $x_j$  dans D
12:    Si l'épisode termine à l'état  $j + 1$  fixer  $y_j = r_j$ 
13:    Sinon,  $y_j = r_j + \gamma * max_a Q_*(s_j, a, \theta_*)$ 
14:    Faire une descente de gradient sur  $(y_j - Q(s_j, a_j, \theta))^2$  selon le paramètre  $\theta$ 
15:    Mettre à jour  $Q_* = Q$ 
16:   end for
17: end for
  
```

4.3.4 • BILAN ET DIFFICULTÉS RENCONTRÉES

Bien que nous ayons compris les aspects théoriques à l'oeuvre dans cet algorithme, nous avons jusqu'à présent rencontré des difficultés à obtenir des résultats exploitables en utilisant celui-ci. En effet, une fois l'apprentissage effectué, nous n'avons pas réussi à dégager une réelle plus-value pour notre méthode par RL, par rapport aux méthodes développées jusqu'à présent. Ceci nous amène donc à nous interroger sur les choix que nous avons faits pour développer celle-ci, peut-être faudrait-il envisager d'implémenter des méthodes profondes plus proches de l'état de l'art et donc plus performantes, notamment celles faisant partie de la classe des méthodes "Acteur-Critique".

5

CONCLUSION

5.1 RÉSULTAT FINAL

L'objectif de ce PSC était **l'identification de chemins allant d'un point A à un point B** à partir de données de vents dans le cadre du guidage d'un ballon atmosphérique dont **seule l'altitude peut être contrôlée**.

Pour aborder ce problème, deux méthodes ont été utilisées, et combinées afin d'obtenir un modèle précis, efficace et rapide.

La première repose sur des **algorithmes traditionnels** qui ont démontré une efficacité notable dans la prévision de trajectoires précises. Sous l'hypothèse d'une connaissance absolue des vents, ces algorithmes nous ont notamment permis de constituer une base de données (de trajets) de taille considérable qui aura été essentielle pour les différents algorithmes de *machine learning*.

La seconde méthode reposait sur des réseaux de neurones entraînés afin de guider le ballon en temps réel. Pour traiter des données aux dimensions atypiques et de grande taille, de nouvelles architectures ont d'abord été développées, reposant sur des **réseaux convolutifs et résiduels** afin d'atteindre les performances des meilleurs algorithmes traditionnels, en utilisant la grande quantité de données qu'il nous était possible de générer. Ainsi, nous sommes parvenus à produire des environnements capables d'apprendre le comportement des algorithmes, et à les reproduire leurs résultats dans le cadre de chemins nouveaux. À partir de ces modèles déjà performants, capables d'extraire les informations nécessaires de données afin de guider convenablement le ballon, nous avons appliqué des méthodes d'apprentissage par renforcement dans l'optique d'obtenir des résultats supérieurs aux premiers algorithmes.

5.2 PISTES D'AMÉLIORATION

Notre projet initial a principalement considéré les vents comme unique contrainte affectant la navigation des ballons stratosphériques. Cette approche, basée sur une hypothèse simplifiée de connaissance absolue des vents, a montré des résultats prometteurs avec des algorithmes traditionnels. Toutefois, la réalité des conditions des vents, souvent imprévisibles, souligne la nécessité d'intégrer des techniques de *Machine Learning*, comme le *Reinforcement Learning*, pour optimiser la navigation. L'obtention d'un trajet valide sous l'hypothèse d'exactitude des vents reste insuffisante en pratique. En effet, si le ballon ne peut pas s'adapter aux dérives imprévues, ces imprécisions s'accumulent au

cours du trajet et le ballon ne rejoindra pas la destination. Une utilisation en continu des algorithmes classiques pourrait être une solution mais le *Reinforcement Learning* et les approches bayésiennes fournissent le cadre naturel pour résoudre ce genre de problématiques.

L'intégration de variables supplémentaires constitue également une piste d'amélioration majeure qui rendrait le *Reinforcement Learning* particulièrement pertinent. En ajoutant des facteurs tels que l'élévation solaire, qui influence la pression et donc les mouvements verticaux du ballon, ainsi que le niveau de batterie, crucial pour les opérations prolongées, le problème devient plus complexe mais nettement plus pertinent pour des applications pratiques. La prise en compte de ces paramètres permettrait de mieux répondre aux défis auxquels fait face Stratolia.

De manière similaire, il est envisageable que certaines régions de l'espace soient inaccessibles, par exemple en raison de conditions météorologiques extrêmes. Dans ce contexte, il serait nécessaire de revoir le traitement de nos données pour correctement prendre en compte les positions relatives des différentes couches d'altitude. Cette adaptation permettrait d'intégrer ces contraintes supplémentaires, susceptibles de survenir lors du guidage de ballons dans des conditions réelles.

Par ailleurs, une amélioration de la précision des mesures des vents et une meilleure gestion des données volumineuses pourraient considérablement accroître l'efficacité des guidages en fournissant des informations plus fidèles à la réalité. De même l'exploitation de l'évolution temporelle des vents par nos modèles reste sous-optimale. En intégrant de manière plus approfondie ces variations, notamment pour les trajectoires de longue durée, nos modèles pourraient bénéficier de prédictions plus robustes.

BIBLIOGRAPHIE

- [1] **GitHub PSC.** Lien du dépôt GitHub de notre PSC pour accéder au code source du projet. URL: https://github.com/Mathixx/PSC_Stratolia.
- [2] Marc G. Bellemare et al. “Autonomous navigation of stratospheric balloons using reinforcement learning”. In: *Nature* (2020). URL: <https://www.nature.com/articles/s41586-020-2939-8>.
- [3] Naiyan Wang Bing Xu, Tianqi Chen, and Mu Li. “Empirical Evaluation of Rectified Activations in Convolutional Network”. In: (2015).
- [4] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: (2015).
- [5] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* (2014).
- [6] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512 . 03385 [cs.CV].
- [7] Hyongsuk Kim et al. “Optimal path finding with space- and time-variant metric weights via multi-layer CNN”. In: (2002). URL: <https://onlinelibrary.wiley.com/doi/10.1002/cta.199>.
- [8] Tomas Kulyvicius et al. “One-Shot Multi-Path Planning Using Fully Convolutional Networks in a Comparison to Other Algorithms”. In: (2021). URL: <https://www.frontiersin.org/articles/10.3389/fnbot.2020.600984/full>.
- [9] Du Tran et al. “A Closer Look at Spatiotemporal Convolutions for Action Recognition”. In: (2018). URL: <https://arxiv.org/pdf/1711.11248v3.pdf>.
- [10] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: (2021). URL: <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>.
- [11] *The Balloon Learning Environment*. URL: <https://research.google/blog/the-balloon-learning-environment/>.
- [12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.
- [13] Christopher J.C.H. Watkins and Peter Dayan. “Q-Learning”. In: (1992).
- [14] Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* (2015).

A

ANNEXE : MAJORIZATION DU NOMBRE DE POINTS EXPLORÉS DANS L'ALGORITHME WIDE-SEARCH

Rappel du contexte : la condition pour sélectionner un point p_0 s'écrit :

$$\forall p \in \mathcal{P}, d(p, p_0) > \sqrt{\epsilon \times d(\text{dest}, p_0)},$$

où d donne la distance entre deux points, ϵ correspond au paramètre de précision et \mathcal{P} est l'ensemble des points sélectionnés.

Résultat. *Le nombre de points sélectionnés sur la surface de la Terre est au plus de l'ordre de 10^4 .*

Démonstration. Cette preuve n'a pas l'intention d'être parfaitement exacte mais seulement de fournir un ordre de grandeur du nombre maximum de points sélectionnés.

À chaque point p sélectionné, nous associons une case $\mathcal{C}(p)$ délimitée par le disque ouvert de centre p et de rayon $d_{\min}(p)/2$ où $d_{\min}(p)$ est la distance de p au point sélectionné le plus proche. On peut vérifier que ces cases sont disjointes.

Soit $p_0, p_1 \in \mathcal{P}$, supposons qu'il existe $x \in \mathcal{C}(p_0) \cap \mathcal{C}(p_1)$. Notons $d_{\min} := \max(d_{\min}(p_0), d_{\min}(p_1))$. On a :

$$d_{\min} \leq d(p_0, p_1) \leq d(p_0, x) + d(p_1, x) < d_{\min}(p_0)/2 + d_{\min}(p_1)/2 \leq d_{\min}$$

où on a utilisé l'inégalité triangulaire. Cette supposition conduit à une absurdité, ce qui prouve bien que les cases sont disjointes.

Sélectionner des points avec la contrainte ci-dessus revient donc à réaliser un pavage de la surface terrestre à partir de ces disques. On peut montrer que le pavage le plus dense suit approximativement le schéma suivant (voir l'image à la page suivante) : pour une distance à la destination donnée, on juxtapose des disques côté à côté jusqu'à faire un tour complet. Ensuite on augmente la distance des centres à la destination et on recommence. Ainsi, le plan est pavé par des disques étapes par étapes, où chaque étape consiste à faire le tour du pavage en cours avec des nouveaux disques de taille légèrement plus grande (*cf.* la condition de sélection des points). On numérote ces étapes de pavage et on note d_n la distance à la destination des points ajoutés à l'étape n .

Le nombre de points ajoutés à l'étape n est minoré par $\frac{2\pi d_n}{\sqrt{\epsilon} d_n}$, car dans le meilleur des cas les disques sont tangents et la distance entre deux centres est de $\sqrt{\epsilon} d_n$. Ensuite, les disques ajoutés ont tous une aire supérieure ou égale à $\frac{\pi}{4} \times \epsilon d_n$ donc l'aire totale couverte par les disques après n étapes de pavage est d'au moins (on estime que les écarts possibles à cette stratégie de pavage ne feraient qu'augmenter encore l'aire couverte par un même nombre de points) :

$$\sum_{i=1}^n \frac{\pi^2 \cdot \sqrt{\epsilon}}{4} d_n^{3/2}.$$

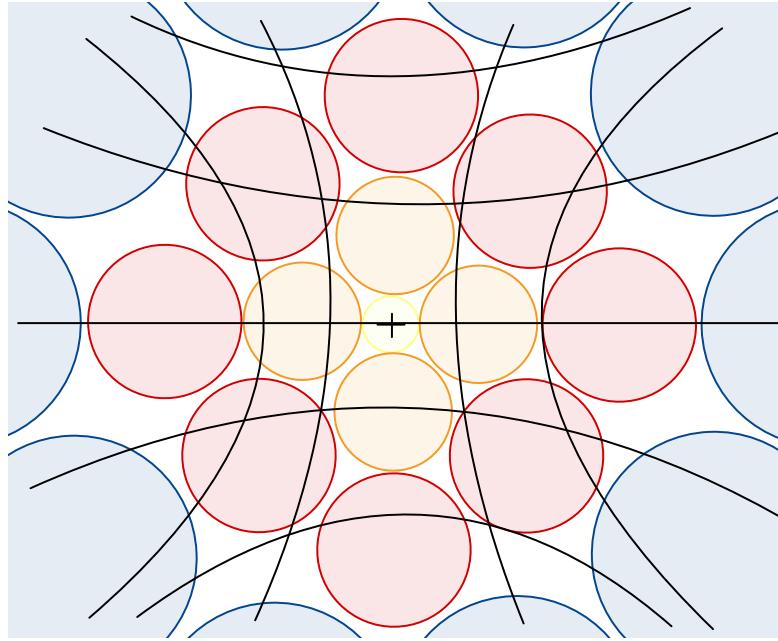


Figure 14: Schématisation de la procédure de pavage

Concernant l'évolution du terme d_n , on peut estimer en étudiant le contact des disques de deux étapes successives qu'il vérifie approximativement la relation $d_{n+1} > d_n + \sqrt{3} \cdot \sqrt{\epsilon d_n}$. Cette relation s'apparente à celle d'une équation différentielle et cette analogie conduit à l'approximation $d_n \approx \frac{3\epsilon}{4} n^2$.

On estime donc par comparaison série-intégrale que l'aire recouverte après n étapes est de l'ordre de :

$$\frac{\pi^2}{2} \epsilon^2 \left(\frac{3}{4}\right)^{3/2} \frac{n^4}{4}$$

Le nombre d'étapes nécessaires pour recouvrir la surface de la Terre est donc majoré par :

$$\left(\frac{8(\frac{4}{3})^{3/2} A_{\text{Terre}}}{\pi^2 \epsilon^2} \right)^{1/4} \approx 51$$

avec A_{Terre} la surface de la Terre. On en déduit finalement le nombre de points ajoutés au total :

$$N \approx \sum_{i=0}^{51} 2\pi \sqrt{\frac{d_n}{\epsilon}} \approx \sum_{i=0}^{51} \pi \sqrt{3} n \approx \boxed{7215}.$$

On obtient donc une majoration de l'ordre de 10^4 (on se limite à une telle borne étant donné les approximations utilisées), conformément à ce qui était annoncé.

□