INF10 RAPPORT FINAL

Algorithmes d'apprentissage pour stabiliser un ballon stratosphérique au-dessus d'une zone bien précise

Rebecca El Chidiac, Imad Barakat, Ouattara Aboubakar, Fritz Morel Epoh Nzoki, Elisa Delbreil

ÉCOLE POLYTECHNIQUE

INSTITUT POLYTECHNIQUE DE PARIS

STRATOLIA

TABLE DES MATIÈRES

1 REMERCIEMENTS

Nous tenons à exprimer notre gratitude envers tous ceux qui ont contribué à la réussite de ce projet. D'emblée, nous remercions chaleureusement la start-up Stratolia, et ses fondateurs Louis Hart-Davis et Hugo Marchand. Ils ont bien voulu nous faire confiance en nous proposant un sujet intéressant et important pour l'évolution de leur entreprise, nous accorder des ressources importantes pour le démarrage de ce projet et nous ont proposé un accompagnement indispensable à la bonne conduite de ce travail.

Nous adressons également notre reconnaissance à monsieur Éric Goubault du département d'informatique de l'École polytechnique, qui a accepté de coordonner la réalisation de ce projet scientifique collectif.

Enfin, nous remercions la scolarité 1A/2A de s'être occupée de la dimension administrative de ce travail, et l'École polytechnique, qui s'investit afin de fournir à ses élèves des enseignements d'excellence à la fois sur les plans technique et humain.

2
EXECUTIVE SUMMARY

Nous avons choisi de réaliser notre PSC avec la startup Stratolia pour plusieurs raisons. Tout d'abord, étant intéressés par le domaine de l'informatique, nous voulions intégrer un projet de manipulation de données afin de confirmer notre attrait pour ce secteur.
Ensuite, nous avons en grande partie axé nos études sur l'informatique et les mathématiques appliquées et étions donc désireux de confronter nos connaissances théoriques à leur mise en application pratique afin de voir si ces disciplines correspondaient à nos attentes.

Durant notre travail sur la problématique de "stationkeeping", nous avons découvert le fonctionnement du Reinforcement Learning et l'implémentation de réseaux de neurones.
De plus, nous avons appréhendé les problématiques liées à la gestion de données quand elles ne sont pas ciblées, voire pas suffisantes.

Au cours du travail sur ce projet, nous avons acquis des compétences aussi bien humaines que techniques. D'une part, nous avons appris à aller vers les autres, à oser poser des questions et à collaborer de façon à être le plus efficace possible.
D'autre part, nous avons développé des compétences techniques comme l'apprentissage de nouvelles bibliothèques de programmation ou l'analyse des résultats. Notre mission a confirmé, pour la plupart d'entre nous, notre intérêt pour le traitement informatique de données.

Nous sommes conscients qu'il reste encore plusieurs pistes d'améliorations des résultats auquels nous sommes arrivés car notre sujet est très vaste et complexe. Cependant, cette expérience nous a permis de nous familiariser avec les questions de Machine Learning et nous a donné les clefs pour une analyse plus fine à l'avenir.

3 INTRODUCTION

Obtenir une imagerie complète et haute résolution de la Terre est très utile dans de nombreux domaines, comme celui de la défense, du BTP ou bien la prévention et la gestion de catastrophes naturelles.

Aujourd'hui, cette imagerie est majoritairement réalisée à l'aide de satellites. Cependant, cette technologie est très coûteuse et produit des images de faible résolution dans le temps et dans l'espace.

L'objectif de la start-up Stratolia, qui est à l'origine de notre sujet, est d'utiliser des ballons stratosphériques pour obtenir des images de la Terre et ainsi s'affranchir des inconvénients des satellites. En effet, les ballons stratosphériques survolent la Terre à une altitude bien plus basse, ce qui permet une résolution spatiale des images inégalée jusqu'alors. De plus, cette technologie permet d'obtenir les images en direct et est beaucoup moins coûteuse.

L'objectif de notre projet est de nous intéresser au stationnement d'un ballon au-dessus d'une zone donnée. En effet, afin d'obtenir des images d'une zone précise, il est important de réussir à garder un ballon au-dessus de celle-ci, en s'éloignant le moins possible. Une telle technologie serait, par exemple, utile pour obtenir des images aériennes des feux de forêt afin de mieux pouvoir les gérer.

Les ballons considérés suivent le champ de vent et changent de direction en se plaçant dans un vent adapté. Pour ce faire, ils modifient leur altitude. En effet, le sens et la direction du vent varient sensiblement avec l'altitude. L'objectif des ballons est de changer d'altitude de façon stratégique afin de suivre les vents qui leur permettent d'évoluer au-dessus de la zone cible.

4
PRÉSENTATION DU PROBLÈME

4.1 Objectifs et données

L'objectif de ce projet est de réaliser un algorithme de *Station Keeping*, qui permette à des ballons stratosphériques à la merci des vents, de stationner au dessus d'une zonne cible. La technologie proposée pour ce travail était le *Reinforcement Learning* (RL) et, à l'origine, sa réalisation pouvait prendre la forme suivante :

- Modélisation de la situation : choix et implémentation de l'environnement, c'est-à-dire des paramètres utiles pour le RL (agent, récompense, espace des actions, des observations...)
- Développement d'algorithmes de RL, visant à entraîner l'agent, dans le cas d'un environnement dans lequel le ballon exploite des données de vents parfaitement connues, sans contrainte sur ses changements d'altitude ;
- Diverses pistes d'approfondissement possibles : introduction des contraintes énergétiques, en limitant le nombre de changements d'altitude possibles par exemple, prise en compte de l'incertitude sur les données de vents, dûe à des prédictions météorologiques imparfaites...

Dès lors, afin de mettre en oeuvre cette démarche et de nous approprier les données, nous sommes partis d'une modélisation très simpliste que nous avons progressivement densifée :

- Tout d'abord, nous avons étudié une situation minimaliste d'un ballon placé dans un cube divisé en 3 couches stratosphériques. Cette première modélisation permet de prendre en main le problème et de découvrir, sur un cas simple, le fonctionnement du RL, avant de complexifier notre modèle. Ensuite, nous avons augmenté le nombre de couches de stratosphère et le nombre de points par couche. De cette façon, on se rapproche davantage de la réalité. Dans cette partie, nous avons utilisé des vents générés aléatoirement et non les données réelles.
- Nous sommes alors passés à une modélisation plus réaliste et avons utilisé les données de vents réelles. Compte tenu de la taille très importante de ces données, nous nous sommes limités à une seule année de vents. On a ainsi eu l'occasion de nous familiariser avec un nouveau format de données, *pickle*. De plus, bien que très nombreuses, ces données ne nous donnaient pas d'information sur certains points de l'espace pour lesquels nous avions besoin de connaître le vent : nous avons alors choisi de les extrapoler par *interpolation*.

Ces différentes étapes de notre modélisation ont été réalisées par *Reinforcement Learning* et plus précisément *Q-learning*.

4.2 Utilisation du Reinforcement Learning

Pour réaliser notre projet, nous avons décidé d'utiliser des algorithmes de Q-learning qui sont basés sur les équations de Bellman.

L'intérêt du Reinforcement Learning réside dans sa capacité à créer des agents capables de naviguer en autonomie dans des environnements très complexes. Le Reinforcement Learning permet de prendre en compte les interactions entre les actions et les récompenses à long terme, ce qui est bénéfique. Dans notre contexte, où les interactions du ballon avec les vents de la stratosphère sont indispensables pour comprendre le principe de navigation, le Reinforcement Learning s'avère particulièrement bénéfique.

De plus, le Reinforcement Learning garantit une adaptabilité précieuse dans la résolution de ce genre de problème. Elle permet au sujet d'apprendre de ses erreurs, en ajustant sa stratégie au fil des réponses de son

environnement, améliorant ainsi ses performances. Ainsi, dans la perspective d'un ballon capable de s'adapter à des données imparfaites et donc à des situations imprévisbles, le Reinforcement Learning est un bon choix. En effet, le RL permet de développer des agents capables de s'adapter à des environnements dynamiques, en constante évolution.
Cette caractéristique est intéressante pour nous, car elle garantit que le ballon puisse naviguer dans différentes zones du globe et pas uniquement dans une zone où le profil du vent est similaire à celui sur lequel il a été entraîné.

Nous allons commencer par présenter les bases théoriques des algorithmes de reinforcement learning.

On se donne un processus de décision Markovien, c'est à dire un 5-uplet (S, A, R, P, ρ_0), avec :
- A l'ensemble des actions
- S l'ensemble des états
- $R : S \times A \times S \longrightarrow \mathbb{R}$ la fonction récompense qui associe une récompense au triplet composé de l'état courant, l'action prise et l'état suivant.
- P la fonction qui à un état et une action associe une distribution de probabilité sur l'ensemble des états, telle que, pour $s' \in S$,
$$P(s,a)(s') = P(s'|s,a) = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a)$$
$P(s'|s,a)$ est la probabilité de passer de l'état s à l'état s', sachant que l'on a pris l'action a.

L'agent suit une politique, qui correspond au choix des actions en fonction des états. Celle-ci peut être déterministe : dans un état donné l'agent fera toujours la même action, ou probabiliste : à chaque état on associe une distribution de probabilité sur les actions. La politique optimale est celle qui va maximiser la récompense associée à la trajectoire, c'est-à-dire la somme des récompenses associées aux états par lesquels passe l'agent $R(\tau) = \sum_{t=0}^{+\infty} \gamma^t R(s_t, a_t, a_{t+1})$.
Le facteur $\gamma \in [0,1]$ permet de favoriser les réponses à court terme. Plus il est proche de zéro, plus on met l'accent sur le fait de se retrouver dans des états favorables au début de l'exploration.

Les algorithmes de Q-learning utilisent la fonction suivante :

$$Q^*(s,a) = \max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi} (R(\tau)|s_0 = s, a_0 = a)$$

Cette fonction renvoie la récompense maximale que l'on peut avoir, en moyenne sur des trajectoires suivant une certaine politique, sachant que l'on commence en s et qu'on fait l'action a. Cela correspond à la moyenne de la récompense obtenue en suivant la politique optimale sachant que l'on commence en s et qu'on fait l'action a.

On a alors, si l'on note $a^*(s)$ l'action prescrite par la politique optimale lorsque l'on est dans l'état s :

$$a^*(s) = \operatorname*{argmax}_a Q^*(s,a)$$

Les algorithmes que l'on va utiliser consistent à initialiser les Q-valeurs Q^* puis à les mettre à jour à chaque itération de l'apprentissage en utilisant l'équation de Bellman.

$$Q^*(s,a) = r(s,a) + \gamma \times \max_{a'} Q^*(s',a')$$

où l'état s' correspond à l'état dans lequel on se retrouve après avoir effectué l'action a en s (on se place dans le cas déterministe, qui est celui de notre problème, où une action en un état donné nous amène dans un unique état).

On réalise ainsi plusieurs itérations qui nous permettent d'entraîner notre agent en mettant à jour les Q-valeurs $Q^*(s,a)$, qui seront stockées soit dans une matrice Q^*, soit dans la dernière couche d'un réseau de neurone.
A l'issue de la phase d'apprentissage, l'agent prendra ses décisions grâce à la formule suivante :
$a^*(s) = \underset{a}{\text{argmax}} Q^*(s,a)$.
En pratique, dans le cas où l'on utilise une matrice, si l'agent est dans l'état n il choisira l'action qui correspond au coefficient maximal de la ligne n de la matrice Q. Dans le cas général, il choisira l'action qui maximise la Q-valeur.

5
PREMIÈRE MODÉLISATION SIMPLIFIÉE

5.1 Modélisation

Avant de commencer à travailler sur les données de vents des datasets ERA5 et NOAA, ce qui représente un volume de données très important, nous avons commencé par nous familiariser avec le problème en écrivant l'algorithme pour un espace restreint avec des données de vents très limitées.
En effet, nous voulions d'abord comprendre le fonctionnement du reinforcement learning, sans avoir à nous préoccuper de toutes les complications inhérentes au fait de travailler sur des données réelles (système de coordonnées compliqué, vent qui varie trop peu dans certaines zones).

Nous avons considéré un espace d'état discret et fini et avons choisi un système de coordonnées très simple, des coordonnées cartésiennes. Nous avons, dans un premier temps ignoré la réelle géométrie du problème, qui complique grandement celui-ci.
Nous nous placés sur un cube, avec, tout d'abord, 27 états possibles (chaque sommet du cube, le milieu des arêtes et le centre du cube). Ce modèle sommaire comporte donc 3 niveaux de stratosphère différents.
Nous avons pris un exemple de vents très simplifié où le vent a la direction des abscisses décroissantes pour le premier et le troisième niveau, et le sens des abscisses croissantes pour le deuxième niveau. Nous avons supposé que le vent était constant au cours du temps.

Nous avons choisi d'imposer un tel vent car, dans ce cas, la stratégie optimale est très simple à comprendre et il était donc plus facile pour nous, dans un premier temps, de voir si notre matrice Q convergeait bien vers le résultat attendu. En effet, l'agent, dans cette situation peut très facilement rester dans la zone en choisissant par exemple, s'il est initialement au milieu du cube, de monter puis descendre et répéter ensuite l'opération.

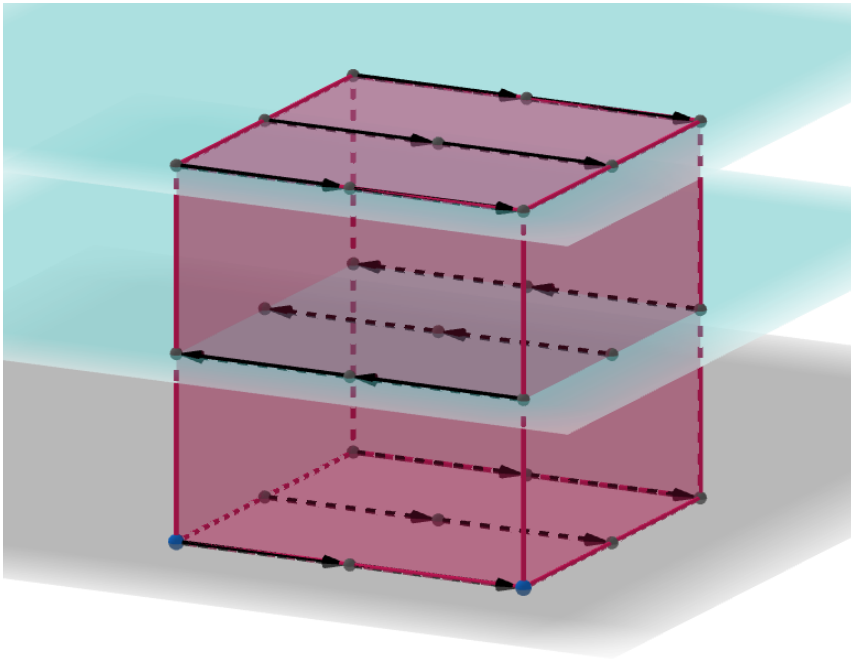


Schéma de la situation, les points sont les états accessibles et les flèches représentent le vent en chaque point.

Le vent correspondant à cette situation est initialisé comme suit :

```
    vents = np.array([[[np.array([0, 0, 0]) for i in range(5)] for j in range(5)] for k in
                                                    range(5)])
for i in range(5) :
    for j in range(5) :
        vents[i][j][0] = np.array([0,-1,0])
        print(i, j)
for i in range(5) :
    for j in range(5) :
        vents[i][j][1] = np.array([1,0,0])
for i in range(5) :
    for j in range(5) :
            vents[i][j][2] = np.array([-1,0,0])
for i in range(5) :
    for j in range(5) :
            vents[i][j][3] = np.array([1,0,0])
for i in range(5) :
    for j in range(5) :
            vents[i][j][4] = np.array([-1,0,0])
```

Nous avons ensuite poursuivi cette modélisation simplifiée en ajoutant des points avec un cube comportant 5 niveaux de stratosphère et 25 points sur chaque niveau, ce qui nous fait $125 = 5^3$ états. En réalité, nous pouvons faire la même chose avec 6^3, 7^3... ce qui a pour effet d'augmenter le temps d'apprentissage.
Nous avons également choisit de façon aléatoire le vent en chacun des points. Par conséquent, la stratégie optimale est plus compliquée à trouver.
Le code que nous présentons est le cas d'un cube $9 \times 9 \times 9$, avec donc 9^3 états. Nous avons utilisé la librairie python Gym, spécialisée dans les algorithmes de reinforcement learning.
Nous commençons par remplir aléatoirement la matrice de vents :

```
import gym
from gym import spaces
import numpy as np
import random
import math
import time

#Initialisation des champs de vent. vents1 pour l'entrainement et vents2 pour la simulation
vents1 = np.array([[[np.array([0, 0, 0]) for i in range(9)] for j in range(9)] for k in range(
                                                    9)])
vents2 = np.array([[[np.array([0, 0, 0]) for i in range(9)] for j in range(9)] for k in range(
                                                    9)])

# Remplissage aleatoire des champs de vents
for i in range(9) :
    for j in range(9) :
        for k in range(9):
            if(random.choice([0, 1])==0):
                vents1[i][j][k] = [random.choice([-1, 1]), 0, 0]
            else:
                vents1[i][j][k] = [0, random.choice([-1, 1]), 0]

for i in range(9) :
    for j in range(9) :
        for k in range(9):
            if(random.choice([0, 1])==0):
                vents2[i][j][k] = [random.choice([-1, 1]), 0, 0]
            else:
                vents2[i][j][k] = [0, random.choice([-1, 1]), 0]
```

On crée ensuite notre environnement et on l'initialise.

```
    # Definition de l'environnement
class VolEnv(gym.Env):
```

```
    def __init__(self):
        super(VolEnv, self).__init__()

        # Definir l'espace d'action - 3 actions : 1 pour 'monter', 2 pour 'descendre', 0
                                                                rester
        self.action_space = spaces.Discrete(3)

        # Definir l'espace discret des etats
        self.observation_space = spaces.Discrete(9**3)  # Pour un cube 9x9x9

        # Etat initial
        self.state = np.array([4, 4, 4], dtype=np.int32)  # Point central du cube

    def step(self, action, vents):

        # Appliquer une action et retourner l'etat suivant, la recompense, et si le processus
                                                            est termine.
        if action == 1:
            self.state[2] = min(self.state[2] + 1, 8)
        elif action == 2:
            self.state[2] = max(self.state[2] - 1, 0)

        # Effet du vent apres avoir execute une action
        self.state += vents[self.state[0]][self.state[1]][self.state[2]]

        # Nouvelle recompense basee sur la distance par rapport au centre
        distance_du_centre = np.linalg.norm(self.state - np.array([4, 4, 4]))
        reward = 1/(1+distance_du_centre)

        # Determiner si on est sorti de la zone ou pas
        done = not ((self.state >= 0).all() and (self.state <= 8).all())


        return self.state, reward, done, {}

    def reset(self):

        # Reinitialiser l'environnement a son etat initial.
        self.state = np.array([4, 4, 4], dtype=np.int32)
        return self.state

# Initialiser l'environnement
Vol = VolEnv()
Vol_test = VolEnv() # Environnement qui choisit des actions de facon equiprobable, pour tester
                                                        le modele
```

On initialise également un deuxième environnement, pour lequel on n'utilisera pas de Q-learning mais on choisira à chaque fois une action de façon équiprobable. Ce sera l'environnement témoin, qui nous servira à évaluer les performances de notre modèle.

Pour la fonction de récompense, on souaite que celle-ci soit haute au centre de la zone et faible lorsque l'on s'éloigne de celle-ci. Nous avons donc opté pour une fonction de la distance au centre de la zone cible, de la forme : $reward = \frac{1}{1+distance}$.

On utilise une stratégie ϵ-greedy, c'est-à-dire que lors de l'entraînement on choisit une action aléatoire avec une probabilité ϵ et, avec probabilité $1-\epsilon$ on prend une action qui maximise la Q-valeur.

```
# Hyperparametres
alpha = 0.1  # Taux d'apprentissage
gamma = 0.6  # Facteur de remise
epsilon = 0.5 # Strategie epsilon-greedy

# Taille de l'espace d'observation et d'action
observation_size = Vol.observation_space.n
```

```
action_size = Vol.action_space.n

# Fonction qui convertit les etats en des nombres pour pouvoir les indexer dans la table de Q
def conversion(s):
    return s[2] + 9 * s[1] + 81 * s[0]

# Fonction qui choisit l'action a prendre
def choose_action(q_table, state):
    if random.uniform(0, 1) < epsilon:
        return Vol.action_space.sample()  # Exploration : choisir une action aleatoire
    else:
        state_c = conversion(state)
        return np.argmax(q_table[state_c, :])  # Meilleure action basee sur Q: Exploitation
    #print(state)
```

On crée ensuite une fonction qui met à jour la matrice des Q-valeurs, selon l'équation de Bellman.

```
# Fonction qui modifie la Q-valeur d'apres l'equation de Bellman
def update_q_table(q_table, state, action, reward, next_state):
    state_c = conversion(state)
    old_value = q_table[state_c, action]
    next_state_c = conversion(next_state)
    next_max = np.max(q_table[next_state_c])
    new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max) #Equation de
                                                    Bellman
    q_table[state_c, action] = new_value

# Fonction qui choisit la meilleur action (celle qui maximise la Q-valeur)
def best_action(q_table, state):
    state_c = conversion(state)
    return np.argmax(q_table[state_c, :])


# Initialiser la table Q
q_table = np.zeros((observation_size, action_size), dtype=np.float32)
```

On passe alors à l'entraînement de l'agent.

```
    # Nombre d'episodes d'entrainement
num_episodes = 10000


# Entrainement de l'agent
for episode in range(num_episodes):
    state = Vol.reset()
    total_reward = 0
    done = False

    while not done:
        action = choose_action(q_table, state)
        current_state = state.copy()
        next_state, reward, done, _ = Vol.step(action, vents1)
        if done:
            break
        update_q_table(q_table, current_state, action, reward, next_state)
        total_reward += reward
```

5.2 Résultats

Présentons tout d'abord les résultats de notre première modélisation des plus simples, où nous avons choisi nous-mêmes le vent. La matrice de Q-valeurs que nous obtenons correspond bien à une politique optimale. En effet, pour chaque état (donc chaque ligne), le coefficient le plus élevé correspond bien à une action qu'il est

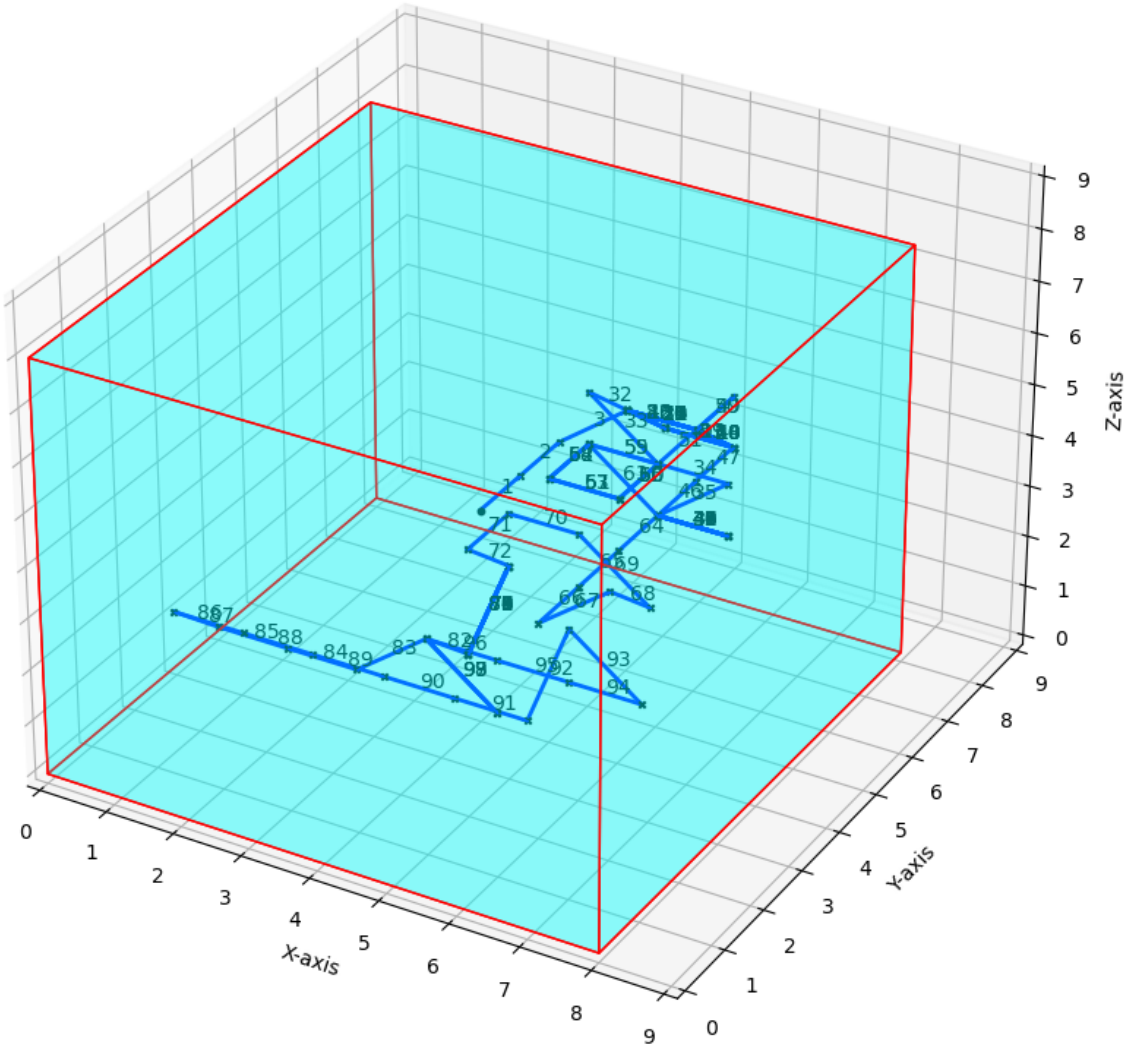
judicieux de choisir. Cette matrice est disponible en Annexe 1.

Présentons maintenant les résultats de notre deuxième modélisation. Après entraînement, on teste notre agent sur un vent différent de celui sur lequel nous l'avons entraîné. De plus, à chaque instant, afin de voir si notre agent arrive à s'adapter, on s'autorise à faire changer totalement le vent, avec une probabilité de 1%.
On réalise 100 étapes (i.e. notre agent choisit 100 actions). On affiche également en direct la trajectoire de notre agent dans le cube à l'aide d'une animation. Le code permettant l'affichage de la trajectoire est disponible en Annexe 2. On fait de même avec l'agent qui agit de façon équiprobable et qui n'a pas été entraîné, afin de comparer les performances.

```
# Simulation d'une trajectoire apres entrainement
for V in [Vol, Vol_test] :
    if V==Vol_test :
        print("Trajectoire du modele equiprobable")
    else :
        print("Trajectoire du modele entraine")
    state = V.reset()
    total_reward = 0
    done = False

    plt.ion()
    cpt = 1

    while cpt<100:
        if V==Vol :
            action = best_action(q_table, state)
        else :
            action = Vol_test.action_space.sample()
        current_state = state.copy()
        state, reward, done, _ = V.step(action, vents2) # Ici les etats evoluent suivent un
                                                        vent aleatoire different de celui de l'
                                                        entrainement
        if(done):
            print(f"Sortie apres {cpt} etapes")
            break
        if V==Vol :
            update_q_table(q_table, current_state, action, reward, state)
        ax.scatter(state[0], state[1], state[2], color='black', marker= 'x', s=10)
        arrow3D(ax, current_state, state, cpt)
        cpt += 1
        plt.pause(0.25)
        total_reward += reward

    # Changement total du champ de vent avec probabilite de 1%
        if(np.random.uniform()<0.05):
            print("Changement de vents")
            for i in range(9) :
                for j in range(9) :
                    for k in range(9):
                        if(random.choice([0, 1])==0):
                            vents2[i][j][k] = [random.choice([-1, 1]), 0, 0]
                        else:
                            vents2[i][j][k] = [0, random.choice([-1, 1]), 0]
    if V==Vol :
        if (cpt == 100):
            print("succes : 100 etapes realisees apres entrainement")
            plt.ioff()
        print(f"Recompense totale apres l'entrainement : {total_reward}")
        # Show the plot
        plt.show()
    else :
        if (cpt == 100):
            print("100 etapes realisees avec modele equiprobable")
        plt.ioff()
```

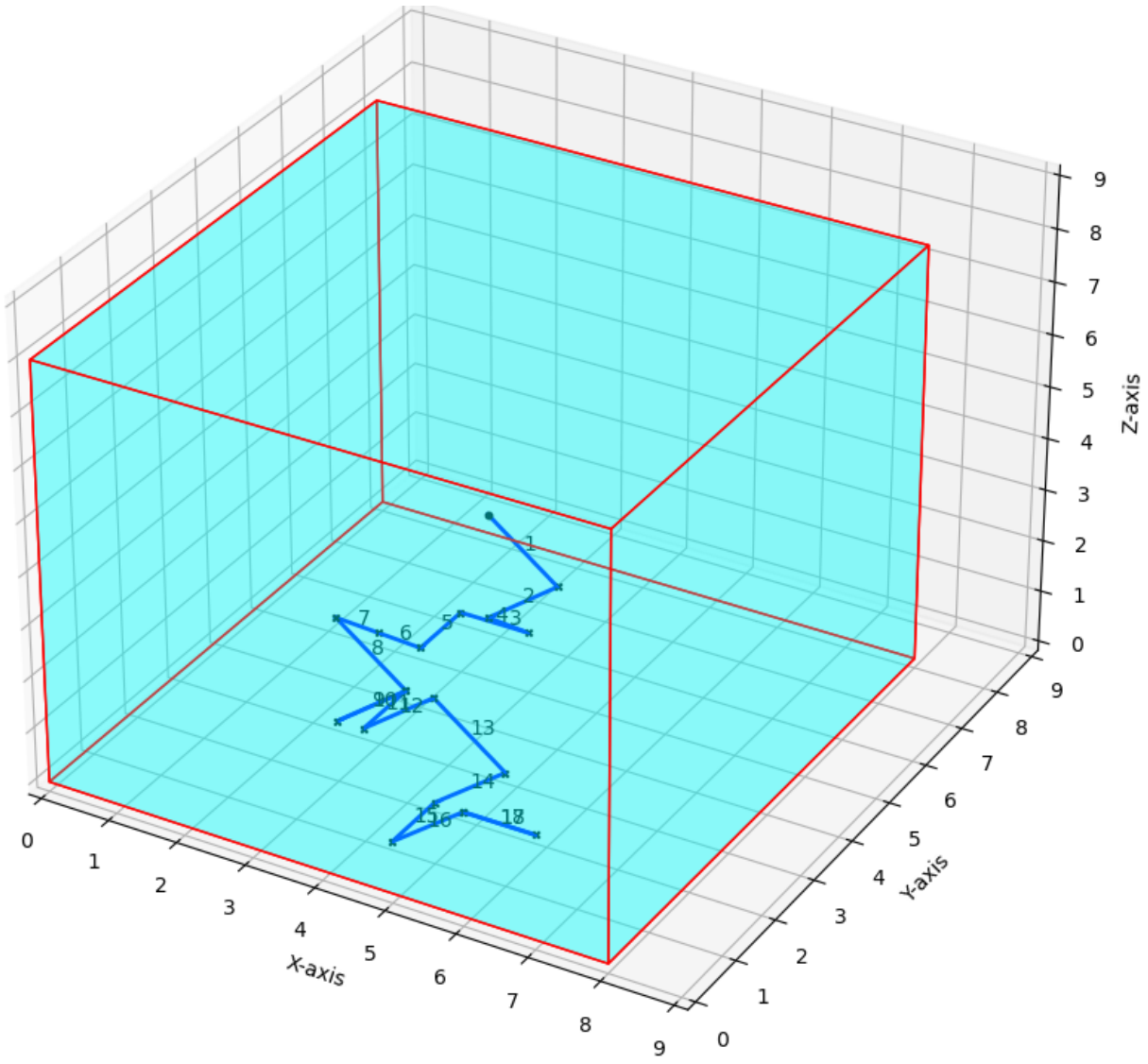
```
        print(f"Recompense totale du modele equiprobable : {total_reward}")
        # Show the plot
        plt.show()
```

On constate que l'apprentissage a eu un effet important, le ballon reste bien dans la zone durant les 100 étapes. Voici un exemple de trajectoire obtenue après avoir exécuté notre code, pour l'agent entraîné :



Exemple de sortie de l'algorithme, qui affiche la trajectoire de l'agent

A titre de comparaison, voici un exemple de trajectoire obtenue après avoir exécuté le code, pour l'agent qui prend des décisions équiprobables. L'agent ne reste jamais dans la zone pendant 100 étapes et sort toujours avant, aux alentours de 30 ou 40 itérations.



Exemple de sortie de l'algorithme, qui affiche la trajectoire de l'agent qui prend des décisions aléatoires

On remarque que pour des vents de petites intensités le ballon s'adapte bien aux changements de vents mais que si l'intensité augmente il sort trop vite pour pouvoir s'explorer car notre cube est trop petit (c'est à dire que l'on a trop peu d'états). On entrevoit ici l'une des limites de notre premier modèle.

5.3 Limites du modèle simplifié

Cette première modélisation n'est pas satisfaisante sur bien des aspects. Le principal problème est que l'on ne travaille pas sur les données réelles. La géométrie utilisée n'est pas du tout représentative de la géométrie de la stratosphère, qui est bien mieux paramétrée en coordonnées sphériques, même si celles-ci sont plus compliquées à manipuler.

De plus, les vents réels ne sont pas aléatoires et ce modèle ne prend évidemment pas en compte la répartition réelle des vents sur le globe. Cette simulation ne nous renseigne donc en rien sur la navigabilité (caractère propice ou non au station-keeping) des différentes zones de la stratosphère.

Nous avons donc par la suite développé des algorithmes capables de travailler sur les données réelles, qui prennent en compte la structure complexe de celles-ci.

Une autre limitation de cette première modélisation est qu'elle ne peut pas être adaptée à un modèle qui comporte un grand nombre d'états, voire un espace d'états continu. En effet, on stocke les Q-valeurs dans une matrice, où le nombre de lignes est égal au nombre d'états.

Afin d'analyser et de manipuler les données réelles, nous aurons besoin de travailler avec un espace d'états continu. De plus, le volume des données ne nous permet pas de continuer avec un algorithme de simple Q-learning. Par conséquent, nous avons développé une deuxième modélisation qui utilise des algorithmes plus évolués, basés sur le deep Q-learning. Les algorithmes de ce type ont l'avantage d'utiliser des réseaux de neurones pour stocker des Q-valeurs, à la place de matrices.

6
DEUXIÈME MODÉLISATION PLUS RÉALISTE

6.1 Modélisation

La première modélisation simpliste nous a permis de visualiser, dans un petit environnement, le comportement de l'agent et les décisions qu'il prend après l'entrainement. Cependant, cette modélisation était beaucoup trop simple pour représenter la réalité car elle ne nous permet pas de travailler avec le sdonnées réelles de vent. Nous avons alors procédé à une deuxième modélisation plus réaliste.

Tout d'abord, il nous a été nécessaire de modéliser le problème, afin d'ensuite choisir les paramètres de nos algorithmes.
L'objectif de notre projet, qui correspond aux besoins de Stratolia, était, dans l'idéal de réussir à maintenir le ballon au-dessus d'une zone sur la surface terrestre de la forme d'un cercle de rayon 50 kilomètres, tout en restant dans la stratosphère.
La deuxième contrainte est la plus simple à modéliser, elle correspond simplement à l'obligation de maintenir la pression dans un certain intervalle.

Au vu des données que nous possédons, qui sont en coordonnées sphériques, nous avons traduit cette contrainte en termes de longitude et de latitude.
Nous avons considéré, étant données les coordonnées d'une zone cible (θ, ϕ), θ étant la latitude et ϕ la longitude, que rester dans la zone correspondait à garder sa latitude dans l'intervalle $[\theta - d\theta; \theta + d\theta]$ et sa longitude dans $[\phi - d\phi; \phi + d\phi]$, pour $d\theta$ et $d\phi$ bien choisis. En effet, les deux problèmes restent très proches, seule la forme de la zone cible change et ce changement est mineur.

Nous avons alors tenté d'évaluer les valeurs de $d\theta$ et $d\phi$, dans le cas d'une zone circulaire de rayon 50 km. L'objectif était d'avoir une idée concrète de la taille, en latitude et en longitude de la zone cible qu'il nous faudrait réussir à atteindre, dans des conditions idéales.
Etant en coordonnées sphériques, nous n'avons pas considéré un cercle mais une petite zone élémentaire, dans laquelle on fait varier longitude et latitude et dont l'aire doit s'approcher de celle du cercle.
Nous avons ensuite calculé la taille de la zone cible.

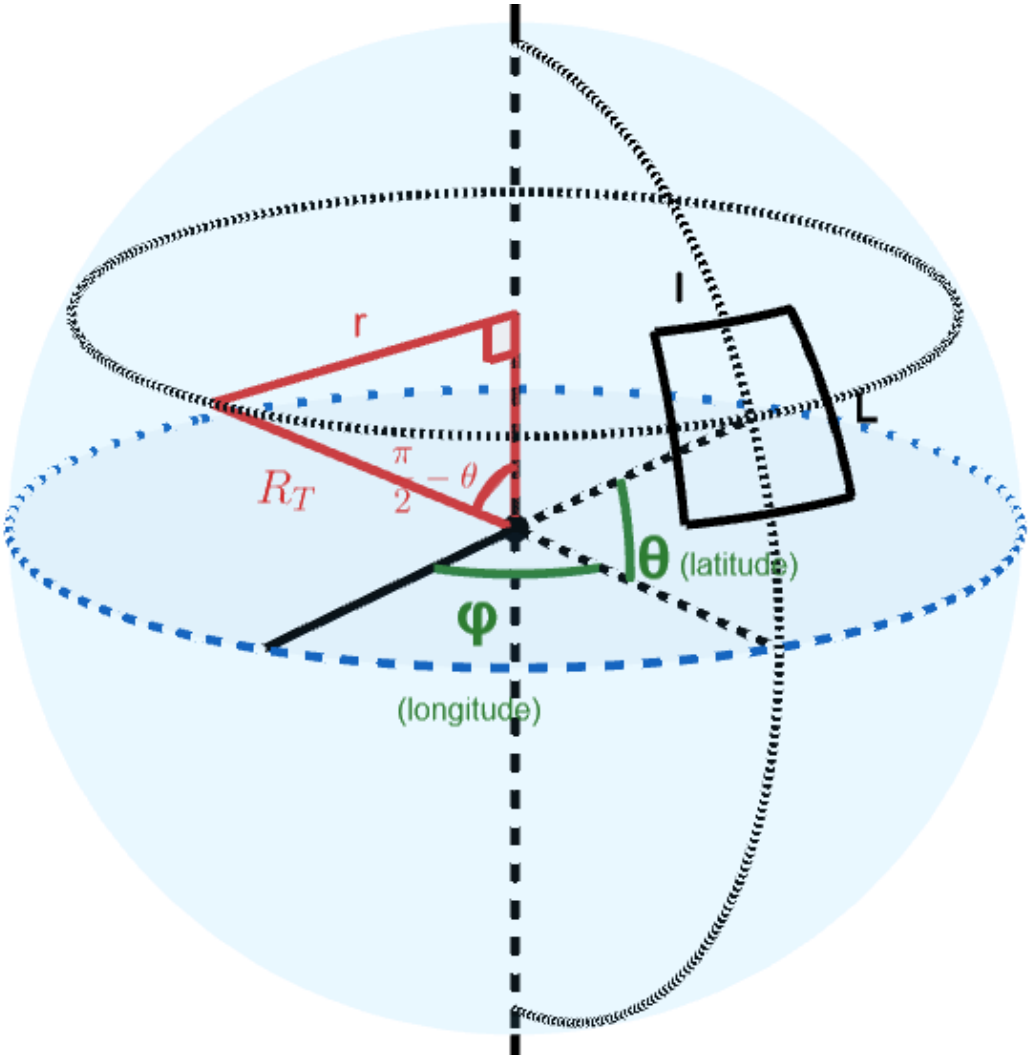


Schéma de la zone cible

En faisant l'approximation raisonnable que l'aire de la zone cible est le produit $l \times L$ (cf schéma ci-dessus), c'est-à-dire en approximant la zone par un rectangle, on peut calculer son aire.

On a : $L = R_T d\theta$ et $l = r d\phi$
De plus, en considérant le triangle rouge, on obtient que : $r = \sin(\frac{\pi}{2} - \theta) R_T = \cos(\theta) R_T$.
On a alors que : $Aire = R_T^2 \cos(\theta) d\phi d\theta$
En première approximation, on décide de choisir $d\phi$ et $d\theta$ tels que : $d\phi = d\theta = \epsilon$.
On a alors : $\epsilon = \sqrt{\frac{Aire}{\cos(\theta) R_T^2}}$
Si l'on considère une zone dont l'aire correspond à celle d'un disque de rayon 50km et que l'on se place au niveau de l'équateur ($\cos(\theta) = 1$), on obtient $\epsilon \cong 0.01$ radians $\cong 0.6°$.

Nous sommes ensuite passés au développement des algorithmes. Pour cela, nous avons utilisé les bibliothèques suivantes, nécessaires pour divers aspects du projet :

```
import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count
import gym
import numpy as np
from scipy.interpolate import interp1d

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

— ‘collections.namedtuple‘ est utilisée pour créer des tuples nommés : ces tuples sont similaires aux tuples standard, mais les champs peuvent être accessibles par des noms de champ, ce qui améliore la lisibilité du code ;
— ‘deque‘ est un conteneur de type file doublement liée qui prend en charge un retrait efficace des éléments à chaque extrémité. On l’utilise pour stocker les expériences de l’agent (c’est-à-dire les suites de type état, nouvel état, action, récompense) dans la mémoire de relecture lors de l’entraînement ;
— ‘itertools.count‘ est une fonction qui crée un itérateur qui renvoie des nombres entiers commençant à un certain nombre et incrémentés d’un pas spécifié. On l’utilise pour générer des indices d’épisodes lors de l’entraînement de l’agent ;
— ‘gym‘ est une bibliothèque qui fournit une interface unifiée pour différents environnements de reinforcement learning. Notre travail s’est très largement basé sur l’usage de cette bibliothèque.
— ‘scipy.interpolate.interp1d‘ est utilisée pour effectuer une interpolation unidimensionnelle ;
— ‘torch‘, ‘torch.nn‘, ‘torch.optim‘, ‘torch.nn.functional‘ sont les composants principaux de la bibliothèque PyTorch, qui est largement utilisée pour le deep learning. ‘torch‘ est le package principal, ‘torch.nn‘ fournit des outils pour construire des réseaux de neurones, ‘torch.optim‘ offre divers optimiseurs pour l’entraînement de réseaux de neurones, et ‘torch.nn.functional‘ fournit des fonctions d’activation et des fonctions de perte couramment utilisées dans les réseaux de neurones.

Juste après, on configure Matplotlib pour un fonctionnement interactif dans un notebook IPython, ce qui facilite la visualisation des graphiques dans l’environnement de développement. Cela sera particulièrement utile lors de l’affichage de graphiques animés ou de la mise à jour en temps réel des tracés lors de la visualisation des résultats.

```
# configurer matplotlib
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

plt.ion()
```

Ensuite, on définit notre zone cible :

```
#Zone cible
low_time_cible= 0
high_time_cible= 0
low_press_cible= 4
high_press_cible= 6
low_long_cible= 36
high_long_cible= 108
low_lat_cible= 18
hight_lat_cible= 54
low_wind_x = -50
high_wind_x = +50
```

```
low_wind_y = -50
high_wind_y = +50
```

C'est la zone au-dessus de laquelle on entraîne l'agent à rester. On s'autorise de petits déplacements en pression, longitude et latitude autour du centre de la zone.
Pour la latitude et la longitude, les valeurs présentes sont à ce stade choisies aléatoirement à titre d'exemple. Si une zone est trop petite ou si les vents dans la zone ne varient pas assez, celle-ci peut s'avérer non-navigable, c'est-à-dire que le ballon ne parviendrait pas à rester au dessus quelques soient les décisions qu'il prend. Pour cela, l'objectif est de se placer dans une zone dont la navigabilité est haute.
Alors que pour la pression, les valeurs affichées sont les indices des vraies valeurs de pression délimitant la stratosphère telles qu'elles sont récupérées dans la base de données que l'on utilise : le dataset NOAA.

Dataset	NOAA - Reanalysis II
Nombre de niveaux de pression	17
Nombre de niveaux de pression dans la tranche [16 - 22km]	3
Détail des niveaux de pression	(hPa) : 1000, 925, 850, 700, 600, 500, 400, 300, 250, 200, 150, 100, 70, 50, 30, 20, 10
Résolution spatiale	2,5 ° en latitude et longitude, soit un point tous les 278 km environ
Résolution temporelle	6 heures
Historique disponible	43 ans
Volume de données pour un niveau de pression et une journée de vents	100 Ko
Volume de données pour une année complète de vents	1,5 Go

TABLE 1 – Détails sur les données de vents NOAA

On définit aussi les pas dans le cas d'une action de montée ou de descente. Ce paramètre peut être optimisé afin d'obtenir de meilleurs résultats, il peut être intéressant de varier plus ou moins de pression, et donc d'altitude, pour avoir accès à des vents divers. Néanmoins, il pourrait également être fixé, à cause de contraintes liées aux capacités de propulsion du ballon.

```
#Pas de montee/descente
pas = 0.5

#Fonctions utiles pour l'environnement

#rayon de la terre en m
RT = 637100.0

#intervalle de temps entre deux positions en s
DT = 0.1

#pas de latitude
dtheta = 2.5
```

```
#pas de longitude
dphi = 2.5

#Cible
cible = np.array([low_time_cible, (high_press_cible + low_press_cible)/2, (high_long_cible +
                                                    low_long_cible)/2, (hight_lat_cible +
                                                    low_lat_cible)/2])
```

On crée également des fonctions qui permettent, à partir d'un indice, de connaître la vraie valeur des paramètres en question. En effet, durant tout notre projet, l'une des difficultés fut de naviguer dans les données qui sont en coordonnées sphériques, en convertissant constamment celles-ci en indices, correspondnat à leurs indices dans le dataset.

```
#creation d'un interpolateur pour la pression car celle-ci n'est pas lineaire en l'indice
liste_indices = [i for i in range(17)]
liste_pressions = [10, 20, 30, 50, 70, 100, 150, 200, 250, 300, 400, 500, 600, 700, 850, 925,
                                                    1000]
interp_func = interp1d(liste_indices, liste_pressions, kind='linear')

#donne la pression d'un point en fonction de son indice (pas forcement entier)
def pression(i) :
    return interp_func(i)

#donne l'altitude d'un point en fonction de son indice
def altitude(i) :
    press = pression(i)
    return 0.3048 * 145366.45 * (1 - (press / 1013.25) ** 0.190284)

#transforme un index de la liste des latitudes en une latitude
def conversion_latitude(i) :
    return i*dtheta

#transforme un index de la liste des longitudes en une longitude
def conversion_longitude(i) :
    return i*dphi

#convertit une latitude en un index
def conversion_index_lat(lat):
    quotient = (lat)/dtheta
    return quotient

#convertit une longitude en un index
def conversion_index_long(long):
    quotient = (long)/dphi
    return quotient
```

On définit une fonction qui calcule la distance entre la position actuelle d'un état donné et l'axe vertical passant par une cible spécifiée.

```
def distance_to_vertical_cible_axis(state):
    # Convertit la 'cible' et l'etat en coordonnees cartesiennes
    lat_c, long_c = conversion_latitude(cible[3]), conversion_longitude(cible[2])
    lat_c_rad, long_c_rad = math.radians(lat_c), math.radians(long_c)
    x_c = RT * math.cos(lat_c_rad) * math.cos(long_c_rad)
    y_c = RT * math.cos(lat_c_rad) * math.sin(long_c_rad)

    # Convertit la latitude et la longitude en radians
    lat_deg = conversion_latitude(state[3])
    long_deg = conversion_longitude(state[2])
    lat_rad = math.radians(lat_deg)
    long_rad = math.radians(long_deg)

    # L'altitude n'affecte pas x et y
    x = RT * math.cos(lat_rad) * math.cos(long_rad)
    y = RT * math.cos(lat_rad) * math.sin(long_rad)
```

```
    # Calcule la distance a l'axe vertical
    distance = math.sqrt((x - x_c)**2 + (y - y_c)**2)
    return distance
```

Elle procède de la façon suivante : Les coordonnées de latitude et de longitude de la cible sont converties en radians, puis utilisées pour calculer les coordonnées x et y de la cible dans un système de coordonnées cartésiennes, en supposant une sphère (la Terre) avec un rayon fixe. On fait de même pour l'état actuel. La distance euclidienne entre les coordonnées de l'état actuel et celles de la cible dans le système de coordonnées cartésiennes est alors calculée à l'aide du théorème de Pythagore puis renvoyée.
Cette fonction nous sera très utile pour calculer la récompense de notre agent, qui est intrinsèquement liée à la distance à l'axe passant par la cible.

6.2 Définition de l'environnement

Passons à la définition de l'environnement. On a choisi de le représenter par une box : c'est un espace d'observations dans OpenAI Gym, qui représente un espace continu avec des bornes spécifiées pour chaque dimension.
Pour ce deuxième modèle, nous avons décidé de rajouter les données de vent au point où se situe le ballon dans l'état. Ce nouveau paramètre correspond au capteur que le ballon possède, qui lui permet de connaître le vent là où il se situe.

```
class env(gym.Env):
  def __init__(self):
        super(env, self).__init__()

        # Definir l'espace d'action - 3 actions : 1 pour 'rester', 2 pour 'monter', 0 pour '
                                                descendre'
        self.action_space = gym.spaces.Discrete(3)

        # Definir l'espace d'etat discret
        self.observation_space = gym.spaces.Box(low =
                        np.array([low_time_cible, low_press_cible, low_long_cible,
                                                                low_lat_cible,
                                                                low_wind_x, low_wind_y]
                                                                ),
                        high = np.array([high_time_cible, high_press_cible, high_long_cible,
                                                                hight_lat_cible,
                                                                high_wind_x,
                                                                high_wind_y]),
                        dtype = np.float32)  # Pour une annee de vents dans tout l'espace

        # etat initial
        wind = interpolator((low_time_cible, (high_press_cible + low_press_cible)/2, (
                                                high_long_cible + low_long_cible)/2, (
                                                hight_lat_cible + low_lat_cible)/2))
        self.state = np.array([low_time_cible, (high_press_cible + low_press_cible)/2, (
                                                high_long_cible + low_long_cible)/2, (
                                                hight_lat_cible + low_lat_cible)/2,
                                                wind[0], wind[1]])  # Point central du
                                                cube

  def step(self, action):
    #On calcul la nouvelle pression apres l'action
    pression = self.state[1] + pas * (action - 1)
    if(pression > 16):
        pression = 16
    self.state[1] = pression
    #On calcule le vent au nouveau point
    alt = altitude(self.state[1])
```

```
    lat = conversion_latitude(self.state[3]) - interpolator((self.state[0], self.state[1],
                                                    self.state[2], self.state[3]))[1]*DT/alt
    while(lat > 180):
      lat -= 180
    while(lat < 0):
      lat += 180

    long = conversion_longitude(self.state[2]) + interpolator((self.state[0], self.state[1],
                                                    self.state[2], self.state[3]))[0]*DT/alt
    while(long > 360):
      long -= 360
    while(long <= 0):
      long += 360

    self.state[2] = conversion_index_long(long)
    self.state[3] = conversion_index_lat(lat)

    wind = interpolator((self.state[0], self.state[1], self.state[2], self.state[3]))
    self.state[4] = wind[0]
    self.state[5] = wind[1]

    dist = distance_to_vertical_cible_axis(self.state)

    #Calculer le reward
    reward = math.exp(-dist/25000)

    # Determiner si on est sorti de la zone ou pas
    done = not ((self.state[1] >= low_press_cible) and (self.state[1] <= high_press_cible) and
                                                     (self.state[2] >= low_long_cible) and (
                                                    self.state[2] <= high_long_cible) and (self
                                                    .state[3] >= low_lat_cible) and (self.state
                                                    [3] <= hight_lat_cible))

    return self.state, reward, done, dist, {}

  def reset(self, time):
    wind = interpolator((time, (high_press_cible + low_press_cible)/2, (high_long_cible +
                                                    low_long_cible)/2, (hight_lat_cible +
                                                    low_lat_cible)/2))
    self.state = np.array([time, (high_press_cible + low_press_cible)/2, (high_long_cible +
                                                    low_long_cible)/2, (hight_lat_cible +
                                                    low_lat_cible)/2, wind[0], wind[1]])
    return self.state

env = env()
```

— 'init' initialise l'état initial en utilisant les valeurs cibles spécifiées et en interpolant (cf. §6.2) les données de vent pour obtenir les composantes du vent. Elle définit aussi l'espace des actions possibles et l'espace des états observables. Dans cet environnement, il y a trois actions possibles : rester, monter ou descendre, définies par gym.spaces.Discrete(3). L'espace des états observables est un espace continu défini par des limites spécifiques (low et high), correspondant à différents paramètres tels que le temps, la pression, la longitude, la latitude, et les composantes x et y du vent.

— 'step' reçoit une action en entrée et met à jour l'état en conséquence (correspond à un pas). Cela implique de modifier la pression en fonction de l'action choisie, de recalculer la latitude et la longitude en fonction des composantes du vent et de l'altitude, de recalculer les composantes du vent en fonction de la nouvelle position, de calculer la distance par rapport à l'axe vertical passant par la cible, et de calculer la récompense en fonction de cette distance. Enfin, il détermine si l'épisode est terminé en vérifiant si l'état est en dehors de la zone cible.

— 'reset' réinitialise l'environnement à l'état initial à partir d'un temps spécifié.

Choix de la récompense (reward) :

La fonction de récompense est calculée en fonction de la distance de l'état actuel par rapport à l'axe vertical

passant par la cible. Cette fonction est conçue pour encourager l'agent à rester aussi proche que possible de l'axe vertical de la cible, ce qui correspond à son objectif principal. Et donc, plus l'état est proche de l'axe vertical de la cible, plus la récompense est élevée.
On définit la fonction de récompense par l'équation suivante :

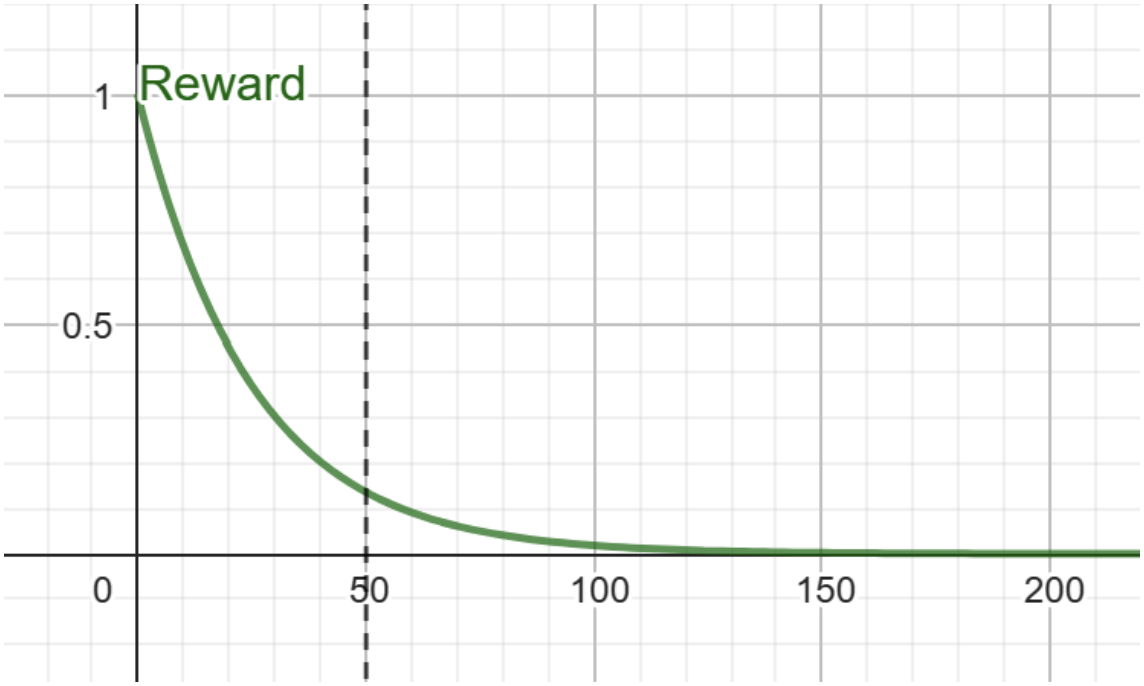
$$R(dist) = e^{-\alpha \cdot dist}$$

où :

- α est un paramètre qui contrôle la pente de la fonction. Plus α est grand, plus la décroissance de la récompense est rapide avec dist
- dist représente la distance calculée à partir de la fonction *distance_to_vertical_cible_axis*().

On ajuste le paramètre α pour contrôler la sensibilité de la fonction de récompense à la distance. Une valeur plus élevée de α rendra la récompense plus sensible aux petites variations de distance, tandis qu'une valeur plus faible de α rendra la récompense moins sensible.

Puisque nous ne voulons avoir une bonne récompense que pour une distance *dist* inférieure à 50 km $= 5 \times 10^4$ m, on choisit $\alpha = \frac{1}{25 \times 10^3}$ m^{-1}.



La récompense en fonction de la distance en Kilomètres
(NB : dans le code, la distance est convertie en mètres)

6.3 Interpolation

Dès que nous avons voulu développer des algorithmes capables d'opérer sur les données réelles dont nous disposions, nous nous sommes rendus compte que la résolution de nos données, qu'elle soit spatiale (en longitude, latitude ou pression) ou temporelle était largement insuffisante.
Par exemple, on dispose d'un point toutes les six heures, ce qui est largement insuffisant pour permettre à notre ballon de prendre des décisions. En effet, il doit prendre une nouvelle décision après quelques secondes de déplacement.

De même, la résolution est de 2.6 degrés pour la longitude, alors que les déplacements qui nous intéressent sont plutôt de l'ordre du centième de degré. En effet, un déplacement de 2.6 degrés en longitude ferait automatiquement sortir le ballon de la zone cible.

Pour des raisons pratiques, nous travaillions avec un dataset contenant les données de vent pour l'ensemble des pressions, latitudes et longitudes possibles sur l'année 2020.
Notre première idée fut de construire au préalable une nouvelle matrice contenant des données plus précises. L'objectif était de choisir le nouveau degré de précision que l'on voulait atteindre sur chacune de nos dimensions, puis de rajouter les points nécessaires. On se fixait donc un nombre de points, pour chaque dimension, à rajouter entre chaque point existant, puis on utilisait l'interpolateur sur ces nouveaux points. On obtenait à la fin de ce processus une nouvelle matrice contenant plus de points.
Cette première solution n'est pas satisfaisante car, au cours de ces déplacements, le ballon, porté par le vent peut se retrouver sur un point qui n'est pas dans le maillage. Il aurait alors fallu créer une autre fonction qui renvoie le point le plus proche pour lequel les données sont stockées dans la matrice. Cette solution aurait amené des complications inutiles.

Nous avons décidé de faire plus simple et de simplement créer l'interpolateur puis de l'évaluer en un point seulement lorsque l'on a besoin du vent en ce point. Cette solution évite de passer trop de temps en amont à évaluer l'interpolateur en des points où ce n'est pas forcément utile.
On a donc créé un interpolateur qui interpole sur chacune des quatre dimensions (temps, pression, latitude, longitude) de façon linéaire.

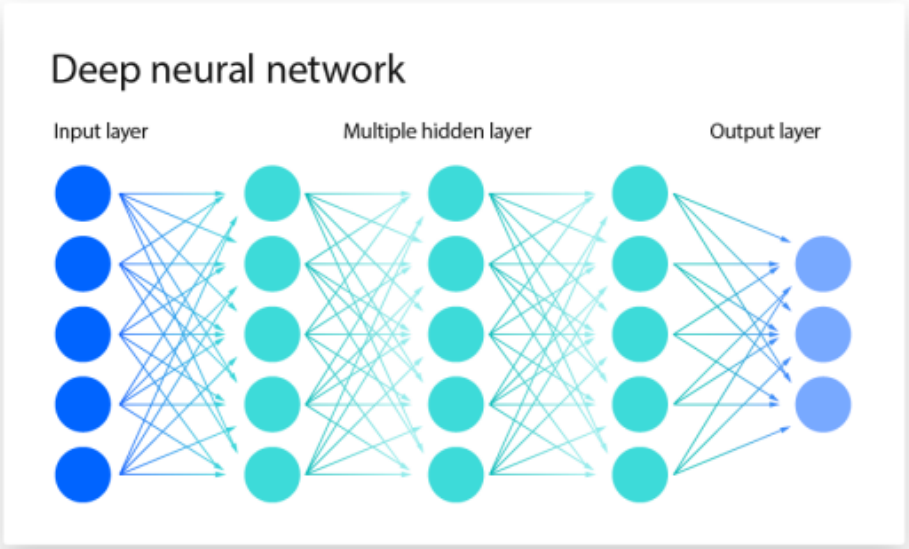
```
#definition de l'interpolateur, construit a partir de la matrice des vents
from scipy.interpolate import RegularGridInterpolator
import pickle

with open("objet_wind_data_2020.pickle", "rb") as f:
    wind_data = pickle.load(f)

vents = wind_data['data']


# Creation de l'interpolateur
interpolator = RegularGridInterpolator((np.arange(1465), np.arange(17), np.arange(144), np.
                                        arange(73)), vents, "linear")
```

6.4 Utilisation du deep Q-learning

Etant donné le grand nombre d'états, il est essentiel de passer d'une Q-valeur tabulaire à un réseau de neurones, capable d'estimer la Q-valeur pour un état donné et une action choisie.

Ceci permet potentiellement de prendre également les vents aux coordonnées de notre agent en considération dans nos états, ce qui permettra de favoriser les états avec un vent qui emmène notre ballon vers la zone. En effet, deux situations où l'agent se trouve en un même point de l'espace mais avec, dans l'une, un vent qui le pousse hors de la zone et dans l'autre, un vent qui le ramène vers le centre ne seront plus considérées de façon équivalente.

Les réseaux de neurones artificiels sont constitués de couches nodales. Ils comprennent une une couche d'entrée, dans notre cas c'est l'état, une ou plusieurs couches cachées et une couche de sortie, qui correspond aux trois Q-valeurs assignées à chaque action. Chaque nœud, ou neurone artificiel, se connecte à un autre et possède un poids et un seuil associés. Si la sortie d'un nœud est supérieure à la valeur de seuil spécifiée, ce nœud est activé et envoie des données à la couche suivante du réseau. Sinon, aucune donnée n'est transmise à la couche suivante du réseau.



Un réseau de neurones

Au fur et à mesure que nous entraînons le modèle, nous voulons évaluer sa précision à l'aide d'une fonction de coût (ou de perte). L'objectif est de réduire au minimum notre fonction de coût pour assurer la précision de l'ajustement pour chaque observation. Au fur et à mesure que le modèle modifie ses poids et biais, il s'appuie sur la fonction de coût et l'apprentissage par renforcement pour atteindre un point de convergence, ou un minimum local. L'algorithme utilise la méthode de la descente de gradient pour ajuster ses poids.

Nous passons maintenant au Deep Q-Learning (DQN), qui a pour but d'estimer la Q-valeur avec un réseau de neurones, et mettre à jour les Q-valeurs en utilisant les équations de Bellman. Le Deep Q-Network (DQN) est une technique d'apprentissage par renforcement qui présente plusieurs avantages clés pour la gestion des environnements complexes, tout en conservant certaines limitations importantes. Voici un aperçu détaillé des avantages associés à l'utilisation du DQN :

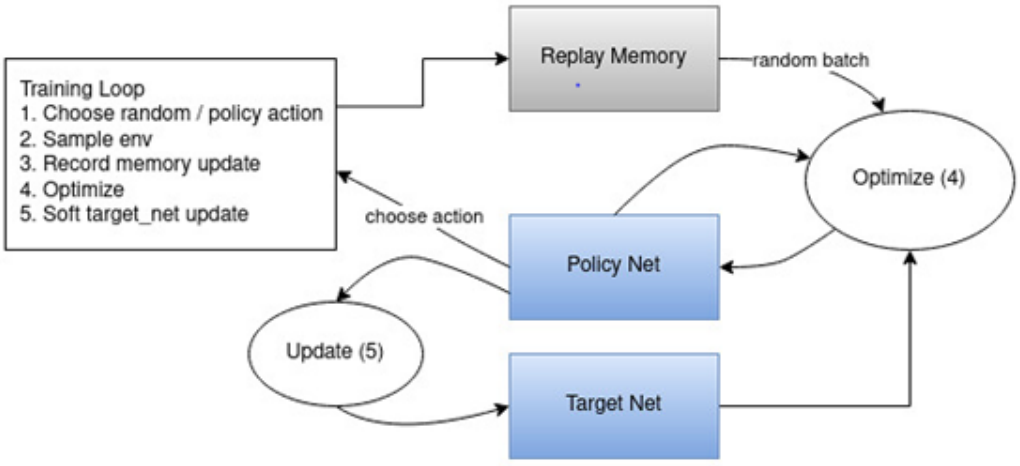
- Apprentissage de représentation profonde : Le DQN utilise des réseaux de neurones profonds pour apprendre des représentations abstraites et de haute dimension des états, permettant un apprentissage efficace dans des environnements complexes.
- Efficacité d'échantillonnage : La relecture d'expérience et les réseaux cibles améliorent l'efficacité d'échantillonnage en réutilisant et en décorrélant les expériences.
- Généralisation : Le DQN peut généraliser les politiques apprises aux états non vus, permettant une meilleure adaptation et prise de décision dans des scénarios nouveaux.

Cependant, le DQN présente également des limitations :

- Sensibilité aux hyperparamètres : La performance du DQN est sensible aux réglages des hyperparamètres, tels que le taux d'apprentissage, le taux d'exploration et l'architecture du réseau, nécessitant un réglage minutieux.
- Manque d'apprentissage continu : Le DQN est principalement conçu pour l'apprentissage par lots hors ligne et ne gère pas naturellement les scénarios d'apprentissage continu en ligne.
- Surévaluation des valeurs d'action : La mise à jour Q-learning utilisée dans le DQN peut conduire à une surévaluation des valeurs d'action, impactant la précision de la politique apprise. Pour assurer la bonne performance de notre réseau DQN, il est nécessaire d'utiliser les notions suivantes :
- Experience replay : il s'agit de stocker dans une certaine "mémoire" les expériences passées, sous forme de transitions, pour les utiliser durant l'entraînement. Ceci permet une utilisation plus efficace de l'expérience

antérieure, en apprenant avec elle plusieurs fois.
— Policy network et Target network : on travaille sur 2 réseaux de neurones. La raison est que notre réseau principal cherche à estimer la Q-valeur, mais cette dernière est mise à jour à chaque étape selon l'équation de Bellman. Ceci engendre donc une instabilité dans l'apprentissage, puisqu'on essaye de rapprocher nos prédictions d'une valeur qui change constamment. On utilise donc un deuxième réseau de neurones, qu'on utilise dans l'équation de Bellman pour calculer la nouvelle Q-valeur, que notre réseau principal cherche à estimer. On parle donc de double DQN, avec un réseau qui est optimisé à chaque étape, et un autre réseau qui sert comme cible et qui est mis à jour après un certain nombre d'étapes.
— Utilisation d'une stratégie ϵ-greedy : cette stratégie offre une balance entre exploration et exploitation. Avec probabilité ϵ, l'agent choisit une action aléatoire (exploration), et avec probabilité $1 - \epsilon$, l'agent choisit la meilleur action, c'est-à-dire celle qui maximise $Q(s, a)$ (exploitation).

On utilise donc l'algorithme suivant :
— Initialiser la mémoire M de capacité N
— Initialiser les réseaux de neurones PolicyNet et TargetNet
— Initialiser l'etat initial s
— Pour épisode i = 1, ... n :
  Pour t= 1, ... T :
  o Choisir une action a en utilisant une stratégie *epsilon*-greedy
  o Executer l'action et observer un reward r et un nouvel état s'
  o Stocker la transition (s, a, s', r) dans M
  o Echantillonner un mini-batch (s, a, s', r) de transitions de M
  o Calculer $Q1(s, a)$ en utilisant PolicyNet
  o Calculer $Q2(s', a)$ en utilisant le TargetNet
  o Calculer $y = r + \gamma * max(Q2(s', a))$ (équation de Bellman)
  o Réaliser une descente de gradient sur $(y \check{} Q1(s, a))$ par rapport aux paramètres de PolicyNet
  o Chaque C étapes effectuer TargetNet = PolicyNet



Algorithme de DQN

Pour l'implémentation, on s'inspire du guide PyTorch. Après avoir défini notre environnement, on procède avec les étapes suivantes :

— On définit le tuple `Transitions` pour pouvoir stocker chaque transition qui est composée de : état, action, nouvel état, récompense.

```
        Transition = namedtuple('Transition',
                                ('state', 'action', 'next_state', 'reward'))
```

— On commence par créer la classe `ReplayMemory`, qui sert de mémoire pour tirer un échantillon aléatoire de transitions pour entraîner le modèle dessus (c'est donc l'experience replay). Par défaut, la taille de cette mémoire est 1000.

```
        class ReplayMemory(object):
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)
    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))
    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)
    def __len__(self):
        return len(self.memory)
```

— On crée la classe `DQN`, qui est notre réseau de neurones dense avec une couche et une fonction d'activation.

```
        class DQN(nn.Module):
  def __init__(self, n_observations, n_actions):
    super(DQN, self).__init__()
    self.layer1 = torch.nn.Linear(n_observations, 300)
    self.layer2 = torch.nn.Linear(300, 600)
    self.layer3 = torch.nn.Linear(600, 300)
    self.layer4 = torch.nn.Linear(300, 150)
    self.layer5 = torch.nn.Linear(150, n_actions)

    self.activation = torch.nn.LeakyReLU()

  def forward(self, x):
    y = self.layer1(x)
    y = self.activation(y)
    y = self.layer2(y)
    y = self.activation(y)
    y = self.layer3(y)
    y = self.activation(y)
    y = self.layer4(y)
    y = self.activation(y)
    y = self.layer5(y)
    return y
```

— Ensuite, on définit nos hyperparamètres, comme le *batch size*, ϵ pour la ϵ-*greedy strategy* (pour le choix des actions), le *learning rate* ainsi que τ, le taux de modification de notre *target network*.

```
# BATCH_SIZE est le nombre de transitions echantillonnees
# GAMMA est le facteur de reduction
# EPS_START est la valeur initiale de epsilon
# EPS_END est la valeur finale de epsilon
# EPS_DECAY controle le taux de decroissance exponentielle de epsilon, plus eleve
                                                  signifie une decroissance plus lente
# TAU est le taux de mise a jour du target network
# LR est le taux d'apprentissage de l'optimiseur ``AdamW``

BATCH_SIZE = 128
GAMMA = 0.99
EPS_START = 0.9
EPS_END = 0.01
EPS_DECAY = 2000
TAU = 0.001
```

```
LR = 5e-4

n_actions = env.action_space.n

state = env.reset(0)
n_observations = len(state)

# On initialise nos reseaux
policy_net = DQN(n_observations, n_actions)
target_net = DQN(n_observations, n_actions)
# Synchronize the target network with the policy network
target_net.load_state_dict(policy_net.state_dict())

# On choisit l'optimiseur
optimizer = optim.AdamW(policy_net.parameters(), lr=LR, amsgrad = True)

# On initialise la memoire
memory = ReplayMemory(10000)

steps_done = 0
```

— La méthode `select_action` choisit avec probabilité ϵ une action aléatoire, et avec probabilité $1 - \epsilon$ la meilleure action (qui maximise $Q(s, a)$).

```
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * math.exp(-1. * steps_done /
                                                    EPS_DECAY)
    epsilons.append(eps_threshold)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # on choisit la meilleure action
            return policy_net(state).max(1).indices.view(1, 1)
    else:
        return torch.tensor([[env.action_space.sample()]], dtype=torch.long)
```

— La fonction `plot_durations` affiche les durées des épisodes au cours de l'entraînement, pour avoir une idée de combien de temps notre agent réussit à passer dans la zone. Le code est disponible est Annexe 3.

— La fonction `optimize_model` est la fonction qui entraîne le réseau de neurones :

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)

    # On transpose le batch
    batch = Transition(*zip(*transitions))

    # On calcule un mask pour les etats qui ne sont pas finaux
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                            batch.next_state)), dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # On calcule Q(s_t, a) - le modele calcule Q(s_t), puis on choisit
    # la colonne des actions prises. Ce sont les actions qui auront et prises
    # pour chaque etat selon le policy net
    state_action_values = policy_net(state_batch).gather(1, action_batch)
```

```
    # Calculer V(s') (c'est a dire max Q(s', a) sur a) pour tous les etats suivants.
    # Les valeurs attendues des actions pour les non_final_next_states sont calculees
    # en se basant sur le reseau cible "plus ancien" ; en selectionnant leur meilleure
                                                    recompense avec max(1).values
    # Ceci est fusionne en fonction du masque, de sorte que nous aurons soit la valeur d
                                                    'etat attendue,
    # soit 0 dans le cas ou l'etat etait final.
    next_state_values = torch.zeros(BATCH_SIZE)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(non_final_next_states).max(1).
                                                        values

    # On calcule les valeurs esperees de Q
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # On calcule la Huber loss
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # On optimize le modele
    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

o On commence par prendre un batch aléatoire de transitions (de taille *batch_size*) de notre mémoire ;
o On transpose ce batch, pour avoir chaque composante (état, état suivant, action, récompense) sous la forme de vecteurs ;
o On les convertit en `torch tensors`, pour pouvoir les insérer dans les réseaux.
On utilise un mask, pour pouvoir distinguer les états finaux (quand on sort de la zone) et les états non finaux.
o On calcule la Q-valeur correspondante pour chaque action, en utilisant le policy network.
o On calcule ensuite les prochains états (non finaux) en utilisant le target network.
o On calcule la Q-valeur en utilisant l'équation de Bellman.
o On évalue l'erreur ("loss") entre la Q-valeur calculée par le policy network, et celle calculée avec le target network.
o `optimizer.zero_grad()` remet les gradients à 0 ;
`loss.backward()` calcule les gradients en utilisant la méthode de backpropagation
o `optimizer.step()` met à jour les poids du réseau.

Enfin, on passe à l'entraînement. On choisit le nombre d'épisodes, puis une action.
On obtient alors une récompense et un nouvel état, qu'on ajoute dans notre mémoire. On optimise ensuite notre modèle et on copie les paramètres de policy network dans ceux de target network.
On utilise le soft update au lieu de mettre à jour le target network après un certain nombre d'étapes, c'est-à-dire que l'on modifie le réseau à chaque étape avec un facteur τ petit, afin d'éviter un changement brusque et donc une instabilité dans l'entrainement.

```
if torch.cuda.is_available():
    num_episodes = 600
else:
    num_episodes = 50

limit = 10000

for i_episode in range(num_episodes):
    # Initialiser l'environnement et obtenir son etat
    state = env.reset(0)
    state = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
    for t in count():
        action = select_action(state)
```

```
        observation, reward, done, dist, _ = env.step(action.item())
        reward = torch.tensor([reward])

        if (done or t>limit):
            next_state = None
        else:
            next_state = torch.tensor(observation, dtype=torch.float32).unsqueeze(0)

        # Stocker la transition dans la memoire
        memory.push(state, action, next_state, reward)
        # Passer a l'etat suivant
        state = next_state

        # Effectuez une etape de l'optimisation (sur le PolicyNet)
        optimize_model()

        # Soft update des poids du reseau cible
        target_net_state_dict = target_net.state_dict()
        policy_net_state_dict = policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key]*TAU +
                                         target_net_state_dict[key]*(1-TAU)
        target_net.load_state_dict(target_net_state_dict)

        if (done or t>limit):
            episode_durations.append(t + 1)
            plot_durations()
            break

print('Complete')
plot_durations(show_result=True)
plt.ioff()
plt.show()
```

6.5 Résultats

L'un des enjeux principaux du station-keeping est de se placer dans une zone favorable à ce type de navigation. En effet, il est nécessaire que le cône de vent soit assez large, afin que le ballon ait accès, en modifiant son altitude, à des vents de directions différentes afin de corriger sa trajectoire.

La difficulté principale du projet est que la zone cible est extrêmement restreinte et donc les vents qui y sont disponibles présentent des directions qui varient peu. Ces conditions rendent l'apprentissage très difficile, car le ballon sort rapidement de la zone.

Il est alors nécessaire d'identifier des zones où la navigabilité est haute, c'est-à-dire où l'on dispose de vents aux directions opposées. En effet, dans certaines zones, effectuer du station-keeping est impossible.

Nous avons donc identifié une zone dans laquelle on dispose de vents de directions opposées. Nous nous sommes attachés à avoir des vents dont la composante x est négative et d'autres pour lesquels elle est positive et de même pour la composante y.

Pour une zone réaliste, nous n'avons pas obtenu de résultat satisfaisant car notre agent ne semble pas apprendre. En effet, lorsque l'on teste notre agent après apprentissage, la distance à la cible augmente linéairement, sans jamais diminuer.
Il semble donc que l'agent ne réussit pas à se rapprocher de la zone cible.

Les explications possibles à ce phénomène sont la faible diversité des directions de vents disponibles, une récom-

pense non optimale ou encore des hyper-paramètres non optimisés. Il est également possible que notre durée d'entraînement soit trop faible et que celui-ci ne soit pas optimal.
Nous avons alors lancé une nouvelle exploration dans une zone plus grande, avec un intervalle de longitude et de latitude de 12° en choisissant un pas de temps $DT = 1s$. Nous avons choisi d'augmenter le pas de temps car la zone est plus grande, afin que le ballon en explore une plus grande partie.
Notre modèle n'est pas performant, on constate qu'il reste dans la zone moins longtemps qu'un modèle équiprobable. Les Q-valeurs sont presque les mêmes pour les 3 actions.
Nous allons donc continuer à l'optimiser pour obtenir de meilleurs résultats.

7 CONCLUSION

Ce projet fut très enrichissant pour nous car il nous permit de découvrir un domaine très intéressant et prometteur dont nous ignorions tout : l'intelligence artificielle et plus précisément le reinforcement learning.

Grâce à notre premier modèle, nous avons compris les bases théoriques du RL, qui repose sur les équations de Bellman.

Notre deuxième modèle nous a permis de nous heurter à la difficulté de travailler sur des données réelles, qui sont par nature imparfaites et compliquées à exploiter.

Nos résultats semblent montrer que les algorithmes de RL sont prometteurs pour résoudre les problèmes de station-keeping. Néanmoins, ceux-ci reposent sur de nombreux paramètres à optimiser afin d'obtenir les meilleures performances possibles. Ainsi, nous devons continuer à ajuster ces paramètres afin de sélectionner ceux qui nous donnent le meilleur résultat possible.

Le sujet auquel nous nous sommes attelés est très vaste et il reste encore certaines pistes d'exploration (intégrer les contraintes énergétiques par exemple).

8 ANNEXES

8.1 Annexe 1 : Q-matrice obtenue dans le cas d'une modélisation simplifiée d'un cube 3x3 et de vents fixés

```
[[0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         1.9508238  0.        ]
 [0.         0.         0.        ]
 [0.         0.         1.9508238 ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [1.4508238  1.5847073  0.        ]
 [0.4214711  0.34583297 1.423883  ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         1.3421608  1.5847073 ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]]
```

Ce résultat nous permit de valider, sur un exemple extrêmement simple, nos algorithmes de Q-learning. En effet, les coefficients non nuls de la matrice correspondent bien aux actions qu'il est judicieux de prendre dans la situation très particulière que nous avons considérée.

8.2 Annexe 2 : code pour afficher la trajectoire de l'agent dans le cube, lors de la modélisation simplifiée

```
    # Visualisation de la trajectoire apres entrainement
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection, Line3DCollection

# Definition du cube
side_length = 4

vertices = [
    [4 - side_length, 4 - side_length, 4 - side_length],
    [4 + side_length, 4 - side_length, 4 - side_length],
    [4 + side_length, 4 + side_length, 4 - side_length],
    [4 - side_length, 4 + side_length, 4 - side_length],
    [4 - side_length, 4 - side_length, 4 + side_length],
    [4 + side_length, 4 - side_length, 4 + side_length],
    [4 + side_length, 4 + side_length, 4 + side_length],
    [4 - side_length, 4 + side_length, 4 + side_length]
]

faces = [
    [vertices[0], vertices[1], vertices[5], vertices[4]],
    [vertices[7], vertices[6], vertices[2], vertices[3]],
    [vertices[0], vertices[4], vertices[7], vertices[3]],
    [vertices[1], vertices[5], vertices[6], vertices[2]],
    [vertices[4], vertices[5], vertices[6], vertices[7]],
    [vertices[0], vertices[1], vertices[2], vertices[3]]
]

fig = plt.figure(figsize=(16,12))
ax = fig.add_subplot(111, projection='3d')

ax.add_collection3d(Poly3DCollection(faces, facecolors='cyan', linewidths=1, edgecolors='r',
                                     alpha=.25))

ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')

ax.set_xlim([0, 9])
ax.set_ylim([0, 9])
ax.set_zlim([0, 9])

# Definition des fleches de trajectoires
def arrow3D(ax, p1, p2, arrow_num):
    arrow_length_ratio = 0.1
    arrow_angle = 20

    # Arrow direction
    d = np.array(p2) - np.array(p1)
    arrow_direction = d / np.linalg.norm(d)

    # Arrowhead points
    arrowhead_length = np.linalg.norm(d) * arrow_length_ratio
    arrowhead_width = arrowhead_length * np.tan(np.radians(arrow_angle))
    arrowhead_points = p2 - arrow_direction * arrowhead_length
```

```
    # Plot the line
    ax.plot([p1[0], p2[0]], [p1[1], p2[1]], [p1[2], p2[2]], color='blue', linewidth=2)

    mid_point = [(p1[i] + p2[i]) / 2 for i in range(len(p1))]
    ax.text(mid_point[0], mid_point[1], mid_point[2], str(arrow_num), color='black')

    # Plot the arrowhead
    arrowhead = Poly3DCollection([[
        (arrowhead_points[0], arrowhead_points[1], arrowhead_points[2]),
        (p2[0], p2[1], p2[2]),
        (p2[0] - arrow_direction[0] * arrowhead_width, p2[1] - arrow_direction[1] *
                                                        arrowhead_width, p2[2] -
                                                        arrow_direction[2] * arrowhead_width)
    ]], color='blue')

    ax.add_collection3d(arrowhead)

# Centre du cube
ax.scatter(4, 4, 4, color='black', marker='o', s=10)
```

8.3 Annexe 3 : Affichages pour le deuxième modèle

```
epsilons = []
episode_durations = []

def plot_durations(show_result=False):
    plt.figure(1)
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    if show_result:
        plt.title('Result')
    else:
        plt.clf()
        plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

    plt.pause(0.001)  # pause a bit so that plots are updated
    if is_ipython:
        if not show_result:
            display.display(plt.gcf())
            display.clear_output(wait=True)
        else:
            display.display(plt.gcf())
```

RÉFÉRENCES

[1] OpenAI website - Introduction to RL

[2] Marc G. Bellemare1, Salvatore Candido, Pablo Samuel Castro, Jun Gong, Marlos C. Machado, Subhodeep Moitra, Sameera S. Ponda3, Ziyu Wang. 2020. Autonomous navigation of stratospheric balloons using reinforcement learning

[3] Pytorch website - Pytorch documentation

[4] Loon website - Loon collection (archives du projet Loon)