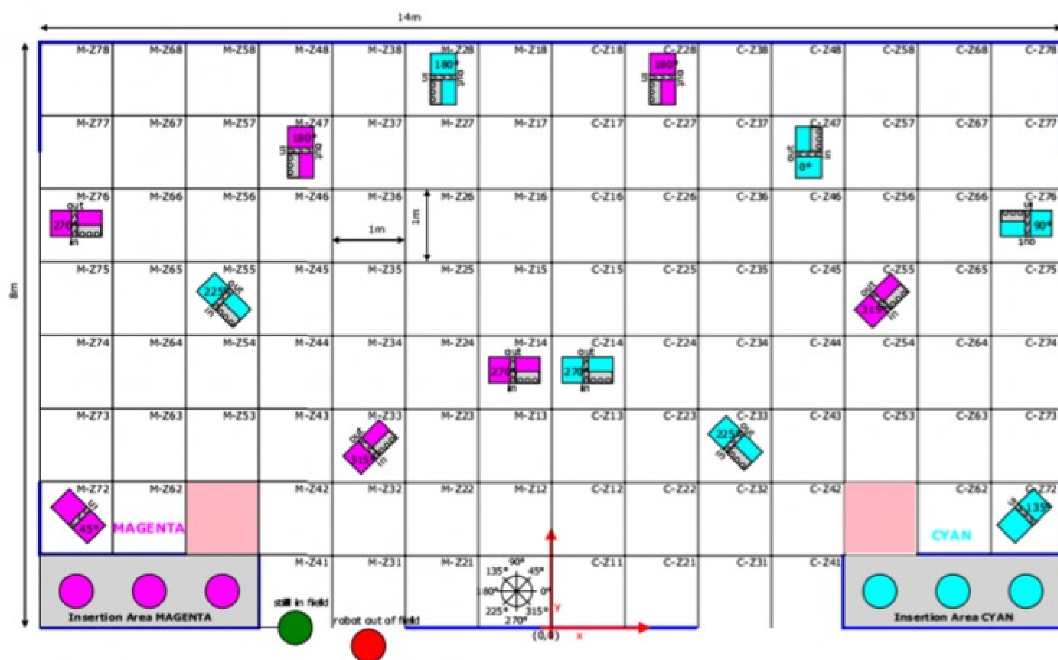
POLYTECH LILLE

Université de Lille 1 SCIENCES ET TECHNOLOGIES

Rapport de projet de 4ème année

Informatique, Microélectronique, Automatique

Machine Learning pour navigation autonome de robots mobiles



Auteur : Wenjing CHEN / Puyun LIN
Encadrants : Vincent Coelen / Andry Zaid Rabenantoandro

Remerciements

Nous tenons à remercier toutes les personnes qui ont apporter leur aide dans le cadre de ce projet :

Monsieur Vincent COELEN et Monsieur Andry Zaid RABENANTOANDRO, encadrants de ce projet, qui nous ont permis de nous lancer dans ce travail. De nombreuses idées ont fusées grâce à nos conversations en début d’année.

Nous tenons également à remercier Monsieur Xavier REDON et Monsieur Alexandre BOE, Un grand merci pour ses temps et ses aide, ils nous ont beaucoup conseillé dans la réalisation du projet.

Table des matières

Remerciements 2

Introduction 4

1 Présentation du projet 5

1.1 Description. 5
1.2 Objectifs 5

2 Etat de l'art 6

2.1 Machine learning. 6
2.2 Réseau de neurones artificiels 6
2.3 Apprentissage par renforcement 7

3 Travail accompli 10

3.1 Collecter des données. 10
3.2 Établir le réseau 11
3.2.1 Prétraitement du modèle 11
3.2.2 Construction d'un modèle d'agent. 13
3.2.2.1 Initialisation. 13
3.2.2.2 CNN. 13
3.2.2.3 Sélection d'action 14
3.2.2.4 Repay 15
3.2.3 Construction d'un modèle d'environement 16
3.2.4 Entraînement 18
3.2.5 Prévision 18
3.2.6 Tracer des trajectoires sur des cartes. 19

4 Démonstration de résultat 21

5 Problème rencontré 25

Conclusion 27

Introduction

Ce projet a été réalisé dans le cadre de la quatrième année de formation Informatique Microéléctronique et Automatique à Polytech Lille.

Notre travail reprend les travaux réalisés par L'Association de Robotique de Polytech Lille. Leur projet consiste l'utilisation d'algorithmes permettant au robot de trouver de manière autonome l' itinéraire le plus court entre le point de départ et la destination. L'objectif de notre travail consiste alors à reprendre cette tâche et à la améliorer avec methode machine learning.

Nous expliquerons tout d'abord le contexte du projet en présentant les objectifs. Nous développerons ensuite les différentes parties de la réalisation du projet. Finalement Nous conclurons en envisageant les ouvertures de ce projet, tout en questionnant sa précision.

1 Présentation du projet

1.1 Description

Aujourd'hui, l'apprentissage automatique (en anglais machine learning) devient de plus en plus la tendance actuelle. Il est maintenant largement utilisée, par exemple l'assistant personnel Siri peut nous aider à envoyer des SMS, à passer des appels, à enregistrer des notes et même à dialoguer avec les utilisateurs. En tant qu'assistant personnel numérique intelligent, Siri utilise la technologie d'apprentissage automatique pour mieux comprendre nos problèmes et demandes de langage naturel. En bref, l'apprentissage automatique est un champ d'étude de l'intelligence artificielle qui se fonde sur des approches statistiques pour donner aux ordinateurs la capacité d' « apprendre » à partir de données, c'est-à-dire d'améliorer leurs performances à résoudre des tâches sans être explicitement programmés pour chacune.

Dans notre projet, l'apprentissage automatique est appliqué au robot. L'objet de notre projrt est que le robot dans un endroit fixe et il peut déterminer une itinéraire optimale par lui-même entre le point de départ et la destination que nous donnons.

Nous allons gérer ce projet par le biais d'une navigation autonome robotique, qui n'inclut ni le positionnement du robot ni les capteurs permettant de détecter les obstacles. La taille et les limites (murs) de la carte que nous utilisons sont fixes, mais la position de l'obstacle change. Nous avons utilisé un réseau de neurone pour identifier l'emplacement des obstacles et l'apprentissage par renforcement pour planifier l'itinéraire optimale.

1.2 Objectif

L'objectif du projet est donc permettre le robot trouver une itinéraire optimale.
Pour ce faire, plusieurs étapes doivent être réaliser :

- Prise de connaissance de machine learning et familiarisation avec l'environnement Tensorflow.
- Prise de connaissance de langage de python.
- Prise de connaissance du principe d'algorithme dans le domaine de machine learning.
- Choisir quelle méthode ou quel réseau dans l'apprentissage automatique dont nous utilisons.
- Etablir notre réseau.
- Tester son précision.

2 Etat de l'art

2.1 Machine learning

L'apprentissage automatique (en anglais machine learning) a eu un grand impact sur la conception de logiciels. Il est très utile parce qu'il nous aide à utiliser des données pour faire la prévision. Mais, quel est différent entre l'apprentissage automatique et la programmation traditionnelle ? Avec les modèles de développement logiciel traditionnels, les programmeurs ont écrit une logique basée sur le problème actuel, puis ont ajouté des données associées. Cependant, la situation change toujours. Et il est impossible pour la programmation traditionnelle d'anticiper les changements. L'apprentissage automatique a l'avantage de nous permettre d'apprendre continuellement à partir de données et de prédire l'avenir. Cet ensemble puissant d'algorithmes et de modèles est utilisé dans toutes les industries pour améliorer les processus et pour mieux comprendre les modèles et les anomalies dans les données.

2.2 Réseau de neurones artificiels

Nous pouvons expliquer la structure des réseaux de neurones artificiels à partir du principe de fonctionnement du cerveau humain. L'élément fondamental d'un réseau de neurones est un neurone. Cette composante de la conscience humaine a les capacités simples : elle peut recevoir des entrées externes. Ensuite beaucoup de neurones sont regroupés d'une certaine manière et effectuent des opérations non linéaires sur les sorties. Enfin ils affichent le résultat final. Nous pouvons exprimer ce processus avec un diagramme de structure de réseau de neurones artificiel. Dans la figure, un petit cercle représente une unité de neurones qui est combinées un réseau complexe selon certaines règles par un entraînement. Ces réseaux effectuent un traitement non linéaire sur la sortie des neurones du niveau inférieur en utilisant différente fonctions mathématiques, Enfin, nous pouvons obtenir les résultats dont nous avons besoin.

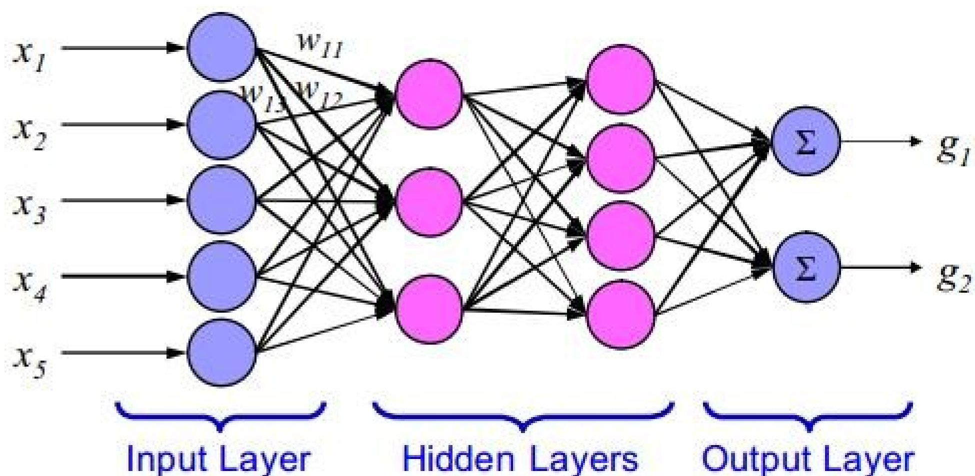


Figure 1 : Réseau de Neurones Artificiels

reconnaissance automatique de la parole, perception des visages, prévision des prix du marché, etc. Dans notre projet, nous avons utilisé le réseau neuronal convolutif (en anglais : Convolutional neural network) pour identifier les obstacles sur la carte.

2.3 Apprentissage par renforcement

L'Apprentissage par renforcement (en anglais : reinforcement learning), une méthode d'apprentissage automatique importante, peut évaluer son processus d'apprentissage lui-même et effectuer une stratégie de récompense et de sanction. Dans le système de l'apprentissage par renforcement, un agent apprend par la méthode essai-erreur, et gagne des récompenses par son interaction avec l'environnement. Son but est de maximiser le profit. L'apprentissage par renforcement peut non seulement réaliser le jeu de labyrinthe, mais aussi prendre également des décisions, par exemple AlphaGo. Nous avons utilisé également un modèle très populaire d'apprentissage par renforcement, « Deep Q-Network » (DQN) pour réaliser la navigation automatique de robot.

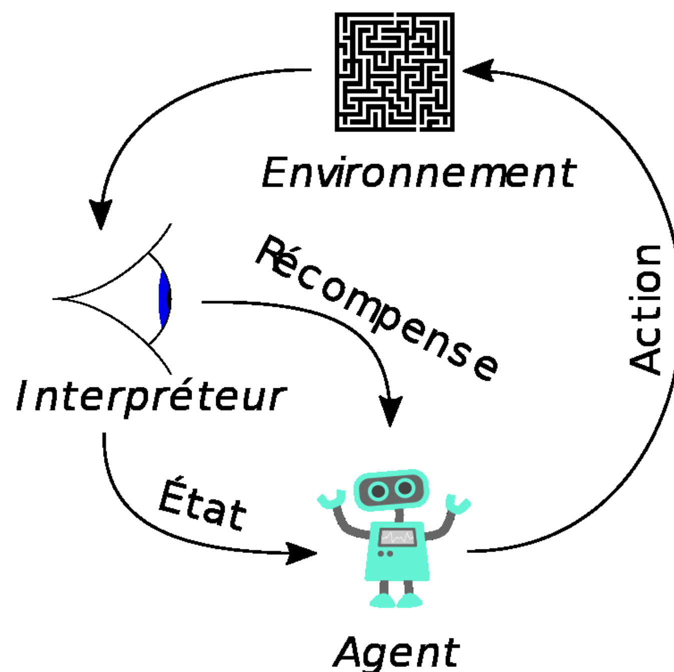


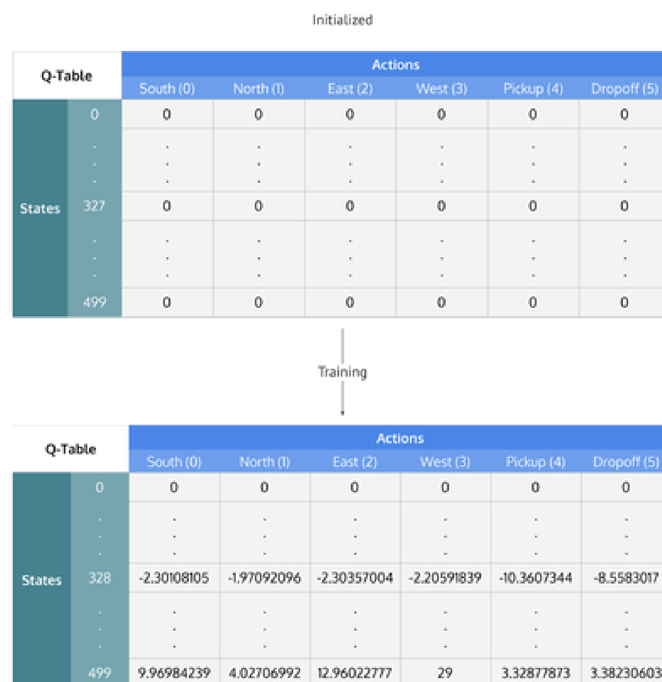
Figure 2 : Apprentissage par renforcement

Ensuite, je veux présenter DQN en détail.

En fait, DQN est une extension de Q-learning. Q-learning est un moyen de l'apprentissage par renforcement. Q-learning est d'enregistrer les règles qui ont été apprises. Les règles informe l'agent quelle action l'agent effectue dans une situation spéciale pour gagner la récompense maximale. Même si une fonction de transfert ou une fonction de récompense ayant des facteurs aléatoires peut être réalisée sans grandes modifications.

La situation consiste en un agent, un ensemble d'état S et d'action A. En réalisant une action a $\in$ A, l'agent passe d'un état à un nouvel état. L'exécution d'une action dans un état spécifique fournit à l'agent une récompense (valeur numérique). Le but de l'agent est de maximiser sa récompense totale. Cela est réalisé par apprentissage de l'action optimale pour chaque état. L'action optimale pour chaque état correspond à celle avec la plus grande récompense sur le long terme. L'algorithme calcule une fonction de valeur action-état : Q = S * A.

Le moyen le plus simple de réaliser Q-learning est de stocker les valeurs de récompense dans un tableau (Q-tableau). Comment obtenir Q-Table ? La réponse est l'initialisation aléatoire, puis obtenir continuellement les informations en retour de l'environnement en effectuant des actions et mettre à jour le tableau via un algorithme. La mise à jour de la valeur Q est une mise à jour de la fonction de valeur.



Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	327	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure 3 : Q-tableau

Lorsque dans un certain état s, l’agent choisit l'action selon la valeur de Q-Table, la récompense obtenue à partir du tableau est Q (s, a). À ce moment, la récompense n'est pas la récompense que nous obtenons réellement, mais le résultat prospectif. Alors où est la vraie récompense ? Nous savons que lorsque nous exécutons l'action a et que nous passons à l'état suivant s', nous pouvons obtenir une récompense immédiate (notée r), mais en plus de la récompense immédiate, nous devons également tenir compte de à la récompense prospective de l'état s' à l’avenir. Donc la récompense réelle (notée Q' (s, a)) se compose de deux parties : récompense immédiate et récompense future. Et la récompense future est souvent incertaine, il est donc nécessaire d’ajouter un facteur d'actualisation γ. Puis la récompense réelle est la suivante :

$$Q'(s,a) = r + \gamma \max Q(s',a')$$

Figure 4 : état actuel de Q

Avec la vraie récompense et la récompense prospective,on peut calculer l'écart entre les deux valeurs et mettre à jour. La valeur mise à jour est le Q (s, a) orginal, les règles de mise à jour sont les suivantes :

$$Q(s,a) = Q(s,a) + \alpha(Q'(s,a) - Q(s,a))$$

Figure 5 : règles de mise à jour de Q

Pour notre projet, l'utilisation de Q-learning seul provoque toujours un problème. Cette approche est limitée par le nombre d'états et d'espaces d'action. Dans notre projet, il y a trop d'états dans le Q-Table, et la Q-Table ne peut pas contenir de mémoire. Nous devons donc modifier Q-learning en utilisant un réseau neuronal convolutif (CNN) qui exprime le Q-Table. C'est la définition de Deep Q-Network (DQN). L'avantage de DQN est qu'il peut résoudre le problème de l'état illimité et de l'action limitée.

La plus grande différence entre DQN et Q-Learning est dans la fonction Experience Replay. En fait, le principe est de répéter des expériences et stocker sample dans memory à l'aide des étapes d'expériences. Chaque étape est un sample qui se compose de : l'état actuel, la valeur Q des différentes actions de l'état actuel, la récompense instantanée obtenu par l'action actuelle et la valeur Q des différentes actions de l'état suivant. Après avoir obtenu un tel sample, vous pouvez mettre à jour le réseau en utilisant l'algorithme de mise à jour Q-Learning mentionné ci-dessus, nous devons effectuer la rétropropagation.

3 Travail accompli

3.1 Collecter des données

La collecte de données est une partie très importante de machine learning car nous avons besoin de données pour former notre réseau. Nous avons utilisé le simulateur que notre encadrant nous a donné. Ce simulateur peut simuler la carte du jeu et simuler de manière aléatoire les obstacles sur la carte. C'est-à-dire que chaque fois que nous redémarrons le simulateur, les positions des obstacles sur la carte seront changé, ces données aléatoires permettent à notre réseau d'être mieux formé.

Ensuite, lorsque le simulateur simule une carte et tous les obstacles, il donne une destination au hasard, et nous devons utiliser la manette pour contrôler le robot afin de l'atteindre. Sur la trajectoire du point de départ à la destination, le système enregistre automatiquement la vitesse angulaire, la vitesse d'abscisse et la vitesse ordinale du robot toutes les 0,01 secondes. En fonction de ces vitesses, nous pouvons tracer la trajectoire emprunté par le robot en python.

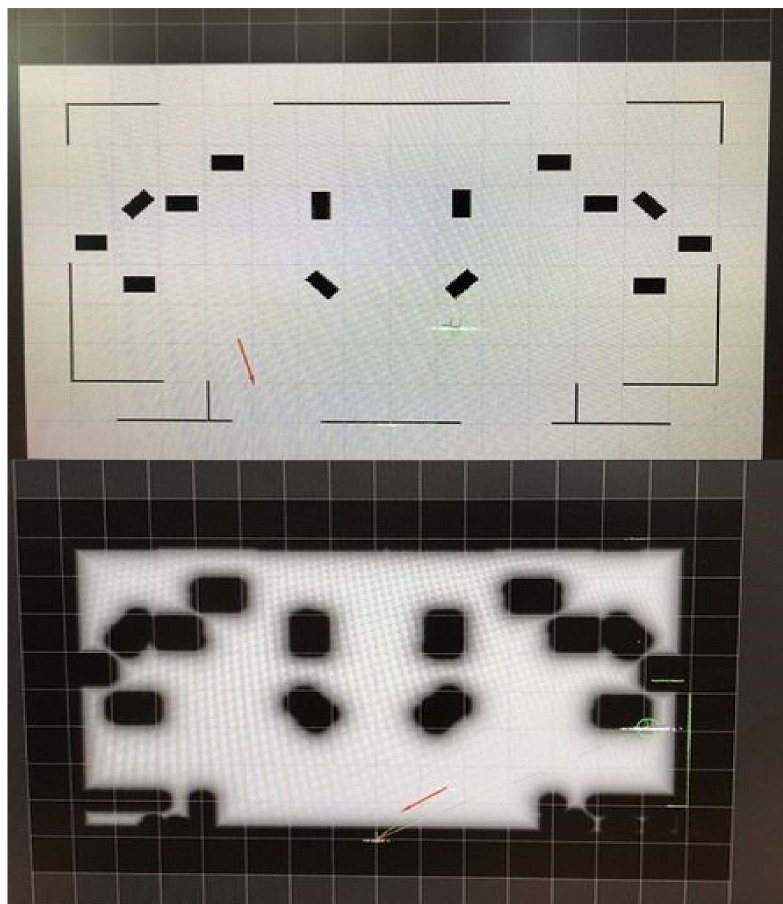


Figure 6 : differentes types de cartes

Quand on ouvre la logicielle rviz, on peut trouver différents types de carte. La première carte simule l'environnement réel. Les rectangles représentent les obstacles. Le cercle représente le Robotino. On peut aussi voir une flèche rouge sur la carte, qui représente le point de destination et sa direction. Alors, il y a l'autre type de la carte. Il nous semble que les obstacles ont les couches épaissies. La fonction de cette couche est d'éviter la collision entre le Robotino et l'obstacle. Parce que l'épaisseur de la couche est supérieure à le rayon du Robotino. On peut aussi trouver que quelques lignes vertes apparaissent sur les obstacles. Ces lignes servent à représenter le contour de l'obstacle mesuré par le capteur de distance. En comparant ces deux types, nous avons finalement choisi le dernier pour collecter des données. Car on n'a pas besoin de s'inquiéter la situation de la collision. Donc, on peut obtenir les données qui est proche de la trajectoire optimale.

Nous avons collecté un total de 60 cartes différentes et 30 trajectoire pour chaque carte (points de départ et d'arrivée sont différents), donc nous avons collecté un total 60*30=1800 trajectoires.

3.2 Établir le réseau

Dans l'introduction précédente, nous avons déjà mentionné que l'apprentissage par renforcement est d'apprendre une stratégie optimale permettant à un agent d'agir dans un environnement spécifique en fonction de l'état actuel pour obtenir le plus de récompense.

Dans notre projet, l'agent est notre robot (robotino) et l'environnement est la carte dans laquelle se trouve le robot. L'idée de base de la navigation autonome est que l'agent effectuera des actions en fonction de la position de départ (avancer, reculer, gauche, droite), tandis que la carte émettra une réaction pour chaque action du robot et que le robot choisira l'action ayant le profit maximal jusqu'à la destination.

Ce que DQN doit faire est de combiner le réseau de CNN avec Q-Learning. L'entrée de CNN correspond aux données des carte originales (en tant que l'état) et la sortie correspond à la fonction de valeur d'évaluation (valeur Q) correspondant à chaque action.

3.2.1 Prétraitement du modèle

Lorsque le robot accède à (ligne, colonne), la valeur de la matrice de la carte en (ligne, colonne) est égale à 2 (*TMP_VALUE*) et en quittant (ligne, colonne), la valeur de la matrice de la carte en (ligne, colonne) retourne à 0. Nous devons également définir le comportement du robot (actions) et nous avons défini quatre actions : avancer, reculer, à gauche et à droite. Ensuite, nous avons également défini les récompenses : si le robot arrive à la destination, il gagne la récompense = 1; si le robot continue à déplacer sur l'espace autorisée de la carte, il est puni -0,01, si le robot sort des limites ou fait la

collision, il est puni -1.

```
TMP_VALUE = 2
start_state_pos = sta_P
target_state_pos = tar_P
actions = dict(
    up = 0,
    down = 1,
    left = 2,
    right = 3
)
action_dimention = len(actions)
reward_dict = {'reward_0': -1, 'reward_1': -0.01, 'reward_2': 1}
```

Figure 7 : Code de définition

Utiliser un ordinateur pour reconnaître une image ne permet pas d'identifier complètement une image complexe à la fois, mais divise une image complète en plusieurs petites morceaux, en extrayant les caractéristiques de chaque petite partie (c'est-à-dire, en identifiant chaque petite morceaux), puis combiner les caractéristiques de ces petits morceaux ensemble, et nous pouvons compléter le processus d'image d'identification de la machine. Donc nous convertissons d'abord la carte en une carte quadrillée en utilisant *find_maze_matrix*, total de 16 x 10 grilles, prenons une valeur tous les 20 pixels pour déterminer si la grille est un obstacle, si c'est un obstacle, la grille sera marquée comme 1, sinon 0, donc puis nous pouvons convertir la carte quadrillée en matrice.

```
def find_maze_matrix(self, input_file):
    img1 = cv2.imread(input_file, 0)
    print(type(img1))
    print(img1.shape)
    r_map = []
    n = 0
    box = None
    for i in range(10, 200, 20):
        for j in range(10, 320, 20):
            print(img1[i, j])
            if img1[i, j] == 0:
                box = 1
            else:
                box = 0

            r_map.append(box)
            n += 1
    print(n)
    r_map_matrix = np.array(r_map)
    r_map_matrix = r_map_matrix.reshape((10,16))
    # r_map_list = list(r_map_matrix)
    print(r_map_matrix)
    return r_map_matrix
```

Figure 8 : Code de find_maze_matrice

Ensuite, nous devons également transforme la matrice de carte précédemment traitée en un tenseur de carte (*matrix_to_img*). Parce que CNN n'accepte pas une matrice d'image comme entrée. Sa shape a trois dimensions : hauteur, largeur et canal de l'image (height,width,channel). Le canal d'image définit la couleur de l'image.Si canal

= 1, il s’agit d’une image en niveaux de gris ; si canal = 3, il s’agit d’une image en couleur RVB. Donc on sélectionne canal = 1.

```
def matrix_to_img(row, col):
    state = copy.deepcopy(maze)
    state[row, col] = TMP_VALUE
    state = np.reshape(state, newshape=(1, state.shape[0], state.shape[1], 1))
    return state
```

Figure 9 : Code de matrice_to_image

3.2.2 Construction d’un modèle d'agent

3.2.2.1 Initialisation

D’abord certains hyperparamètres sont définis et le conteneur de mémoire (*self.meory*) doit être défini pour stocker l’état actuel, l’action, la valeur du récompense, l’état suivant et la variable indiquant que la navigation se termine(*self, current_state, action, reward, next_state, done*).

```
class DQNAgent:
    def __init__(self, agent_model=None):
        self.memory = deque(maxlen=100)
        self.alpha = 0.01
        self.gamma = 0.9  # decay rate
        self.epsilon = 1
        self.epsilon_min = 0.2
        self.epsilon_decay = 0.995
        #
        self.learning_rate = 0.001
        if agent_model is None:
            self.model = self.dqn_model()
        else:
            self.model = agent_model
```

Figure 10 : Code d’initialisation d’agent

3.2.2.2 CNN

Son entrée est le tenseur de carte (vient de la fonction *matrix_to_img*), et la sortie est *current_state, action, reward, next_state, done*. CNN est utilisé pour remplacer Q-table de Q-learning. Il a six couches : une couche d'entrée, une couche de sortie et quatre couches cachées. Dans les trois premières couches cachées sont les couches de convolution à deux dimensions (*Conv2D*) afin de traiter les tenseurs. Les paramètres importants sont suivants :

1. filters : le nombre de noyaux de convolution (c'est-à-dire la dimension de la sortie).
2. kernel_size : taille du noyau de convolution .
3. strides : le pas de la convolution, qui peut être vue comme la longueur du mouvement de convolution .
4. padding : la stratégie du complément « 0 »,et "same" est de conserver le résultat de la convolution au bout, ce qui provoque généralement la même forme en shape de sortie que la forme en shape d'entrée .

La quatrième couche est la couche de développement. Il transforme le tenseur en forme monodimensionnelle (*flatten*). Donc la couche cachée peut être connectée au réseau entièrement connecté (c'est-à-dire la couche de sortie).

```
def dqn_model(self):
    inputs = Input(shape=(maze.shape[0], maze.shape[1], 1))
    layer1 = Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), padding='same')(inputs)

    layer2 = Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), padding='same')(layer1)
    layer3 = Conv2D(filters=128, kernel_size=(3, 3), strides=(1, 1), padding='same')(layer2)
    layer4 = Flatten()(layer3)
    predictions = Dense(action_dimention, activation='softmax')(layer4)
    model = Model(inputs=inputs, outputs=predictions)
    model.compile(optimizer='sgd',
                  loss=mean_squared_error,
                  )
    return model
```

Figure 11 : Code de CNN

```
def flatten(input_list):
    output_list = []
    while True:
        if input_list == []:
            break
        for index, i in enumerate(input_list):
            if type(i)== list:
                input_list = i + input_list[index+1:]
                break
            else:
                output_list.append(i)
                input_list.pop(index)
                break
    return output_list
```

Figure 12 : Code de flatten

3.2.2.3 Sélection d'action

On choisit une action aléatoire pour exploration. La base de l'exploration est le seuil d'exploration de l'action (*self.epsilon*): si *self.epsilon = 0*, l'action peut être sélectionnée aléatoirement, si *0 <self.epsilon <1*, l'action doit être basée sur l'état

actuel. Si l'action la plus possible existe plusieurs directions à la fois, on dérange et choisir une aléatoirement. Car beaucoup de nombre de la valeur maximale de pd.Series peut apparaître , et *argmax ()* ne prendre que la première. Donc On doit utiliser *shuffle* dans bibliothèque sklear pour les déranger.

```
def choose_action(self, state):
    if np.random.rand() < self.epsilon:
        action = random.choice(list(actions.keys()))
        action = actions.get(action)
        return action
    else:
        act_values = self.model.predict(state)
        action = np.argmax(shuffle(pd.Series(act_values[0])))
        return action
```

Figure 13 : Code de choose_action

3.2.2.4 Repay

Ce réseau est instable lorsqu'un réseau de neurones est utilisé à la place d'une table Q pour se rapprocher de la valeur Q avec une fonction non linéaire. Par conséquent, nous devons trouver quelques astuces pour l'aider à converger. À ce stade, nous avons utilisé *Experience Replay*. Pendant la formation, toutes les expériences <s, a, r, s '> sont stockées dans le conteneur de mémoire. Lors de la formation du réseau, à la fin de chaque fois de entrainement, des données de petit lot aléatoires provenant de ce mémoire sont utilisées pour une nouvelle entrainement. Cela rompt la similarité des échantillons d'entraînement ultérieurs et évite la possibilité de minimiser la minimisation locale du réseau. *Experience Replay* rendra également la tâche de formation plus semblable à un apprentissage supervisé ordinaire, simplifiant le débogage et le test de l'algorithme.

La fonction de « Experience Replay » (*repay*) est d'entraîner le modèle: sélectionner au hasard *current_state, action, reward, next_state, done* dans le conteneur de mémoire (*self.memory)*, puis les envoyer au modèle pour y être entraîné.

Dans la boucle : lire les données dans *self.memory*, et définir le nombre d'échantillons lors de l'initialisation. Et puis retirer une partie d'échantillons de manière aléatoire dans le pool de mémoire. *Target_f* est le résultat prévu par CNN. Les données actuelles (*current_state, action, reward, next_state, done*) sont enregistrées dans chaque échantillon, afin de calculer la valeur Q cible de ces données d'échantillon, il faut d'abord retirer le "*next_state*" de l'échantillon (remarque : voici c'est la liste des "*next_state*"). Envoyer le "*next_state*" en tant que paramètre au réseau de neurones, obtenir une table de valeurs Q de toutes les actions exécutables lorsque l'agent est dans l'état *next_state* qui représente l'ensemble des tableaux de valeurs Q pour tous les états *next_state* de l'échantillon.

Maintenant, on a déjà obtenu l'état actuel de l'agent (*current_state*), l'action à ce moment-là, la récompense obtenue en exécutant l'action dans l'état actuel (*reward*),

Valeur exécutable de Q pour l'état suivant.

Nous pouvons maintenant utiliser la formule mentionnée ci-dessus pour calculer la valeur Q cible Q (état, action) : target = reward + Gamma * Max {q_next}.

Selon l'état, si l'action actuel sélectionnée est une action non-exécutable, il ne fait pas de calcul pour l'expérience, et il est puni directement déduit le score, sinon utiliser la formule ci-dessus pour calculer Q (état, action). Enregistrer le Q de tous les échantillons calculés dans *target_f*. L'ensemble d'échantillons de l'état actuel, l'ensemble des actions sont transmis au réseau de neurones pour entraîner. S'il est arrivé à la destination, *target=reward*.

```
def repay(self, batch_size):
    batch_size = min(batch_size, len(self.memory))
    batch_random_choice = np.random.choice(len(self.memory),batch_size)
    for i in batch_random_choice:
        current_state, action, reward, next_state, done = self.memory[i]
        target_f = self.model.predict(current_state)
        if done:
            target = reward
        else:
            target = reward + self.alpha * (self.gamma * np.max(self.model.predict(next_state)[0]) - target_f[0][action])
        target_f[0][action] = target
        self.model.fit(current_state, target_f, nb_epoch=2, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon = self.epsilon * self.epsilon_decay
        else:
            self.epsilon = self.epsilon_min
```

Figure 14 : Code de repay

3.2.3 Construction d'un modèle d'environement

Après la réalisation de l'agent , nous créons notre environement. Il a trois fonctions principales:

1. Obtenir les actions de l'agent.

2. Faire une réponse environnementale.

3. Retourner de récompense.

Autrement dit, il retourne *next_state*, *reward, done* en fonction de *current_state* et *action.*

Lorsque cela est *done = False*, le jeu n'est pas encore terminé. Lorsque le robot monte, le nombre de lignes -1. À l'inverse, lorsqu'il descend, le nombre de lignes +1. De la même façon, quand il passe à gauche, le nombre de colonnes -1. Si il passe à droite, le nombre de colonnes +1.

Puis, nous avons donné une récompense aux trois situations différentes du robot :

1. Si le robot se positionne sur la position « 1 », c'est-à-dire sur la position de l'obstacle, ou si le robot sort des limites, on retourne *reword_0*.

```
if next_state_pos[0] < 0 or next_state_pos[0] >= maze.shape[0] or next_state_pos[1] < 0 or next_state_pos[1] >= maze.shape[1] \
        or maze[next_state_pos[0], next_state_pos[1]] == 1:
    next_state = copy.deepcopy(current_state)
    reward = reward_dict.get('reward_0')
```

Figure 15 : Code de situation 1 de robot

2. Si le robot passe à la position « 0 », c'est-à-dire que la position n'a pas d'obstacle, mais le jeu n'est pas terminé, on retourne *reword_1*.

```
else:
    next_state = matrix_to_img(next_state_pos[0], next_state_pos[1])
    reward = reward_dict.get('reward_1')
```

Figure 16 : Code de situation 2 de robot

3. Quand le robot arrive à la destination, le jeu termine, alors on retourne *reword_2*.

```
elif next_state_pos == target_state_pos:
    next_state = matrix_to_img(target_state_pos[0],target_state_pos[1])
    reward = reward_dict.get('reward_2')
    done = True
```

Figure 17 : Code de situation 3 de robot

Voici Code complète pour cette partie :

```
class Environ:
    def __init__(self):
        pass
    def step(self, current_state, action):
        row, col = np.argwhere(current_state == TMP_VALUE)[0,1:3]
        done = False
        if action == actions.get('up'):
            next_state_pos = (row - 1, col)
        elif action == actions.get('down'):
            next_state_pos = (row + 1, col)
        elif action == actions.get('left'):
            next_state_pos = (row, col - 1)
        else:
            next_state_pos = (row, col + 1)
        if next_state_pos[0] < 0 or next_state_pos[0] >= maze.shape[0] or next_state_pos[1] < 0 or next_state_pos[1] >= maze.shape[1]
                or maze[next_state_pos[0], next_state_pos[1]] == 1:
            next_state = copy.deepcopy(current_state)
            reward = reward_dict.get('reward_0')
            # done = True
        elif next_state_pos == target_state_pos:
            next_state = matrix_to_img(target_state_pos[0],target_state_pos[1])
            reward = reward_dict.get('reward_2')
            done = True
        else:  # maze[next_state[0],next_state[1]] == 0
            next_state = matrix_to_img(next_state_pos[0], next_state_pos[1])
            reward = reward_dict.get('reward_1')
        return next_state, reward, done
```

Figure 18 : Code de environement

3.2.4 Entraînement

Si le modèle existe déjà, nous chargeons le modèle et l'environnement, puis nous définissons un nombre d'itérations égal à 2000. Nous avons initialement décidé que le nombre d'itérations était égal à 1000, mais nous avons constaté que la précision n'était pas particulièrement élevée. Ensuite, nous augmentons lentement le nombre d'itérations et, finalement, lorsqu'il atteint 2000, la précision est acceptable. Puis, après chaque début d'entraînement, tous les paramètres d'état sont réinitialisés. Une fois le jeu démarré, nous sélectionnons d'abord le comportement, puis nous laissons l'environnement l'implémenter pour promouvoir le processus de jeu et nous nous souvenons du comportement de l'état actuel, du prochain état et de la valeur du bonus. Si le jeu est terminé, quittez le processus et passez à la prochaine itération.

```
def train():
    if os.path.exists(model_name):
        agent_model = load_model(model_name)
        agent = DQNAgent(agent_model=agent_model)
    else:
        agent = DQNAgent()
    environ = Environ()
    episodes = 2000
    for e in range(episodes):
        current_state = matrix_to_img(start_state_pos[0],start_state_pos[1])

        i = 0
        while(True):
            i = i + 1
            action = agent.choose_action(current_state)
            next_state, reward, done= environ.step(current_state,action)
            agent.remember(current_state, action, reward, next_state, done)
            if done:
                print("episode: {}, step used: {}" .format(e, i))
                break

            current_state = copy.deepcopy(next_state)
            if i % 100 == 0:
                agent.repay(100)
        if (e+1) % 200 == 0:
            agent.model.save(model_name)
```

Figure 18 : Code d'entainement

3.2.5 Prévision

Tout d'abord, nous chargeons le modèle et spécifions que nous ne pouvons prendre que 100 étapes au maximum. Dans cette boucle en 100 étapes, nous sélectionnons d'abord l'action et laissons l'environnement à l'action; si le jeu est terminé, il saute hors de la boucle. Enfin, laissez l'état suivant devenir le nouvel état actuel.

```python
def predict():
    pos_set = [start_state_pos]
    actions_new = dict(zip(actions.values(),actions.keys()))
    agent_model = load_model(model_name)
    environ = Environ()
    current_state = matrix_to_img(start_state_pos[0], start_state_pos[1])
    for i in range(100):
        action = agent_model.predict(current_state)
        action = np.argmax(action[0])
        next_state, reward, done = environ.step(current_state, action)
        print('current_state: {}, action: {}, next_state: {}'.format(np.argwhere(current_state==TMP_VALUE)[0,1:3], actions_new[action], np.argwhere(next_state==TMP_VALUE)[0,1:3]))
        next_pos = (np.argwhere(next_state==TMP_VALUE)[0,1:3][0],np.argwhere(next_state==TMP_VALUE)[0,1:3][1])
        pos_set.append(next_pos)
        if done:
            break
        current_state = next_state
    print(pos_set)
    return pos_set
```

Figure 19 : Code de prevision

3.2.6 Tracer des trajectoires sur une carte

Après établir notre réseau, nous stockons les coordonnées(x, y) de tous les points prévus sur une trajectoire dans les fichiers txt (*filename et filenamey*). *orgina_filex, orgin_filex* sont d'indiquer les points de trajectoires donnée par le simulateur. *im* lit la carte.

```python
posx = load_data(filenamex)
posy = load_data(filenamey)
origin_px = load_data(origin_filex)
origin_py = load_data(origin_filey)

im = Image.open("pos/map_000000001.png")
```

Figure 20 : Code de stockage de tajectoire prévu

Pendant le prétraitement des données d'images, la couche de protection sur la carte doit être unifiée : les pixels sont fixés en même valeur (la valeur est 255) afin de supprimer la couche. Ensuite, nous convertissons l'image en matrice d'image, puis nous utilisons des *pandas* pour la traiter. Les points de position prévus sont marqués, où les pixels représentent différentes couleurs en tant que marques. Lorsque le traitement des marques se termine, nous transformons la matrice d'image en les données de type *array*.

```
img = np.array(im)
data = pd.DataFrame(img) #use pandas handle the data

for i in range(0,200):
    for j in range(0,320):
        if data.at[i,j] != 0:
            data.at[i,j] = 255


origin_color = 128
predi_color = 192

for i in range(0,len(origin_px)):
    raw = (int(200-(origin_py[i]-0.5)*20))# 0~200
    col = (int((origin_px[i]+7.5)*20))#make - to + 0~320
    data.at[raw,col] = origin_color


for i in range(0,len(posx)):
    raw = (int(200-(posy[i]-0.5)*20))# 0~200
    col = (int((posx[i]+7.5)*20))#make - to + 0~320
    data.at[raw,col] = predi_color
```

Figure 21 : Code de supression de la couche de protection des obstacles

Ensuite, les cartes données par simulateur sont renversées. Donc nous devrions échanger les valeurs au début et à la fin de la matrice pour renverser l'image à normal.

```
for i in range(0,100):
    for j in range(0,320):
        data.ix[i,j],data.ix[200-1-i,j] = data.ix[200-1-i,j],data.ix[i,j]
```

Figure 22 : Code d'inversion de la carte

Enfin, afin de distinguer les deux trajectoires différentes, nous devons utiliser différentes couleurs pour les représenter. Nous avons utilisé *colormap* pour les intervalles de pixels différents sont colorés différemment. Nous définissons : la trajectoire prévue est bleu et la trajectoire du simulateur est rouge.

```
colors = ['black','gray','red','blue','white'] #grad protect red origin blue predi
cmap = matplotlib.colors.ListedColormap(colors,'indexed')
#0~255 if u divided colors into 3 0->black 255/2->red 255->white  if u divieded colors into 4 the same ...
```

Figure 23 : Code de définition des couleur de différents trajectoires

4 Démonstration de résultat

Premièrement, nous voulons vérifier que si le modèle peut réaliser une naviguation n'importe où sur la carte, nous avons effectué trois groupes de tests : nous utilisons la même carte pour chaque groupe, nous entraînons d'abord une trajectoire sur la carte, puis nous entrons le point de départ et la destination au hasard, une trajectoire doit être prédit directement sans entraînement. Si nous pouvons naviguer sur une trajectoire raisonnable, nous continuerons à changer le point de départ et la destination ; sinon, nous entraînons le même point de départ et la même destination, puis nous entrons le nouveau point de départ et la destination nouvelle pour faire des prédictions.

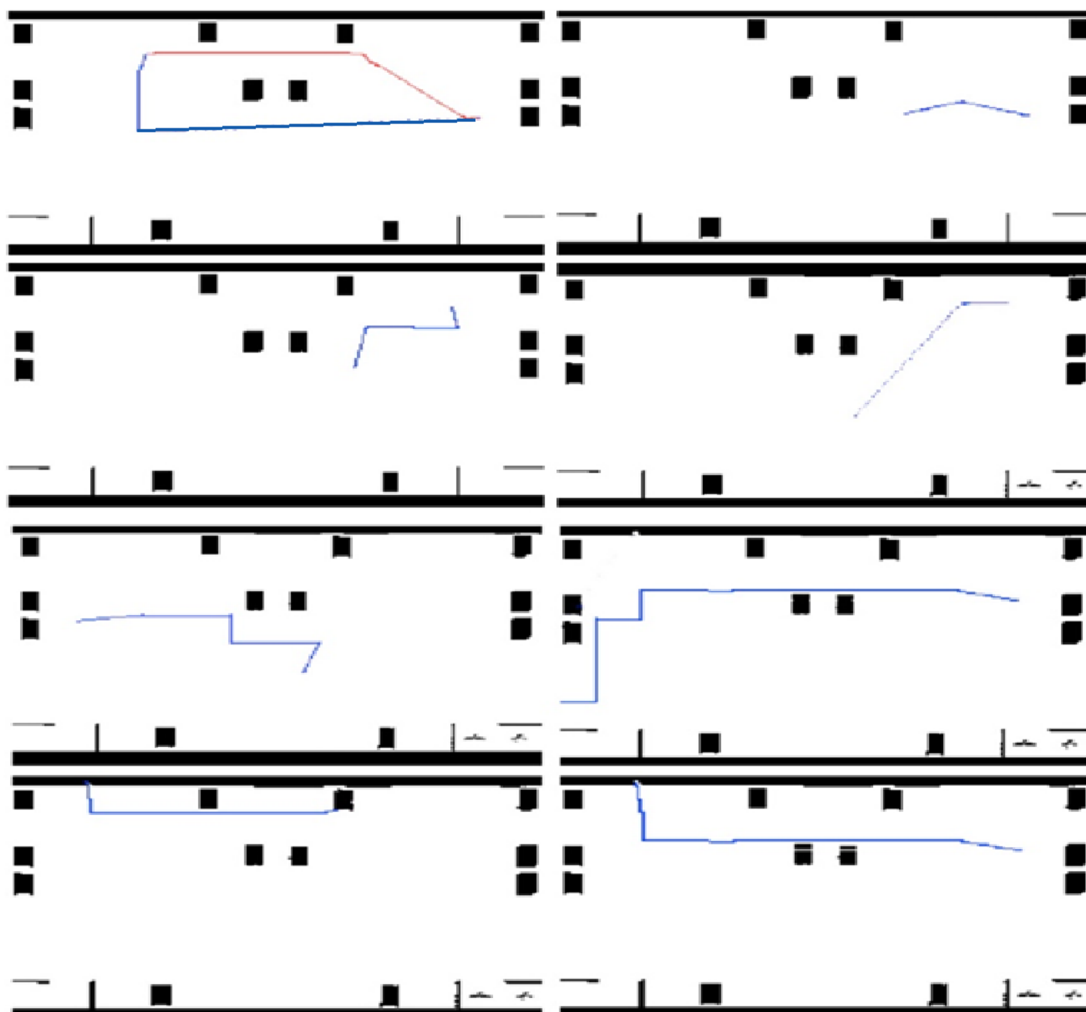


Figure 24 : Première groupe de tester

Le résultat du test est le suivant : le premier groupe, après une fois d'entraînement, le modèle peut donner une trajectoire raisonnable pendant les sept prochaines fois :

Dans le deuxième groupe, après une fois d'entraînement, nous avons donné un nouveau point de départ et une destination, mais le résultat n'était pas idéaux : il avait tendance à aller trop loin. Donc, après nous entraînons encore fois, les cinq prochaines prédictions ont bien fonctionné.

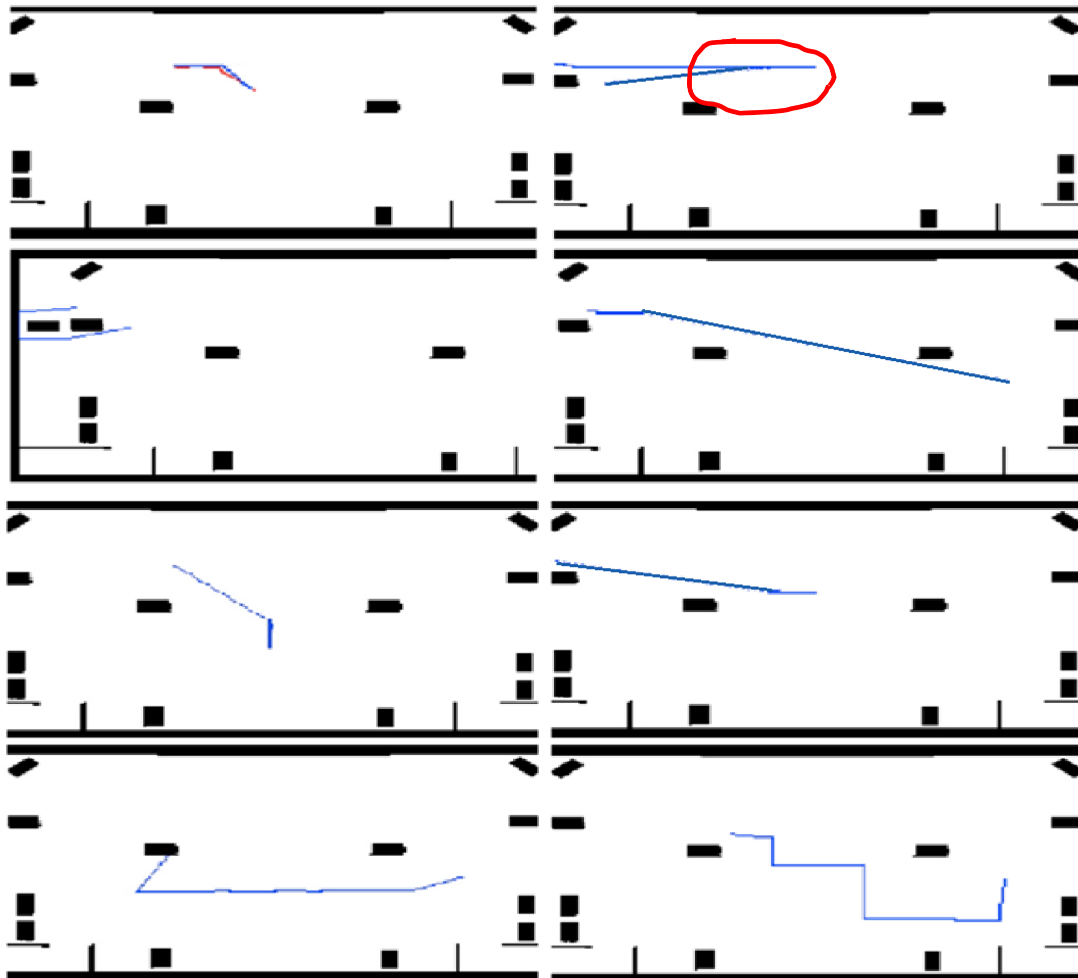


Figure 25 : Douxième groupe de tester

Dans le troisième groupe, après l'entraînement, les sept prévisions suivantes peuvent donner des trajectoires raisonnables.

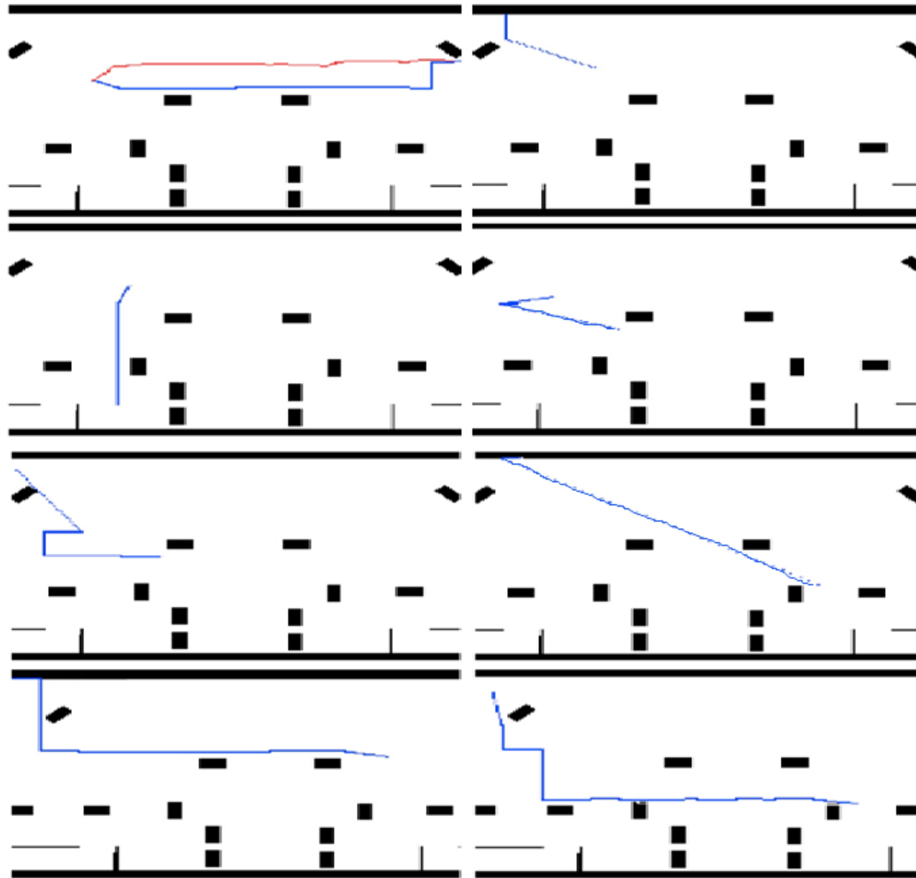


Figure 26 : Troisième groupe de tester

Ensuite, nous devons vérifier si le modèle peut identifier des obstacles sur différentes cartes. Nous donnons donc 16 cartes différentes sans entraînement. Nous avons donné les points de départ et de destination au hasard pour voir le résultat. Le résultat est que le modèle peut planifier des trajectoires évitant des obstacles, mais les trajectoires n'étaient pas toujour optimales.

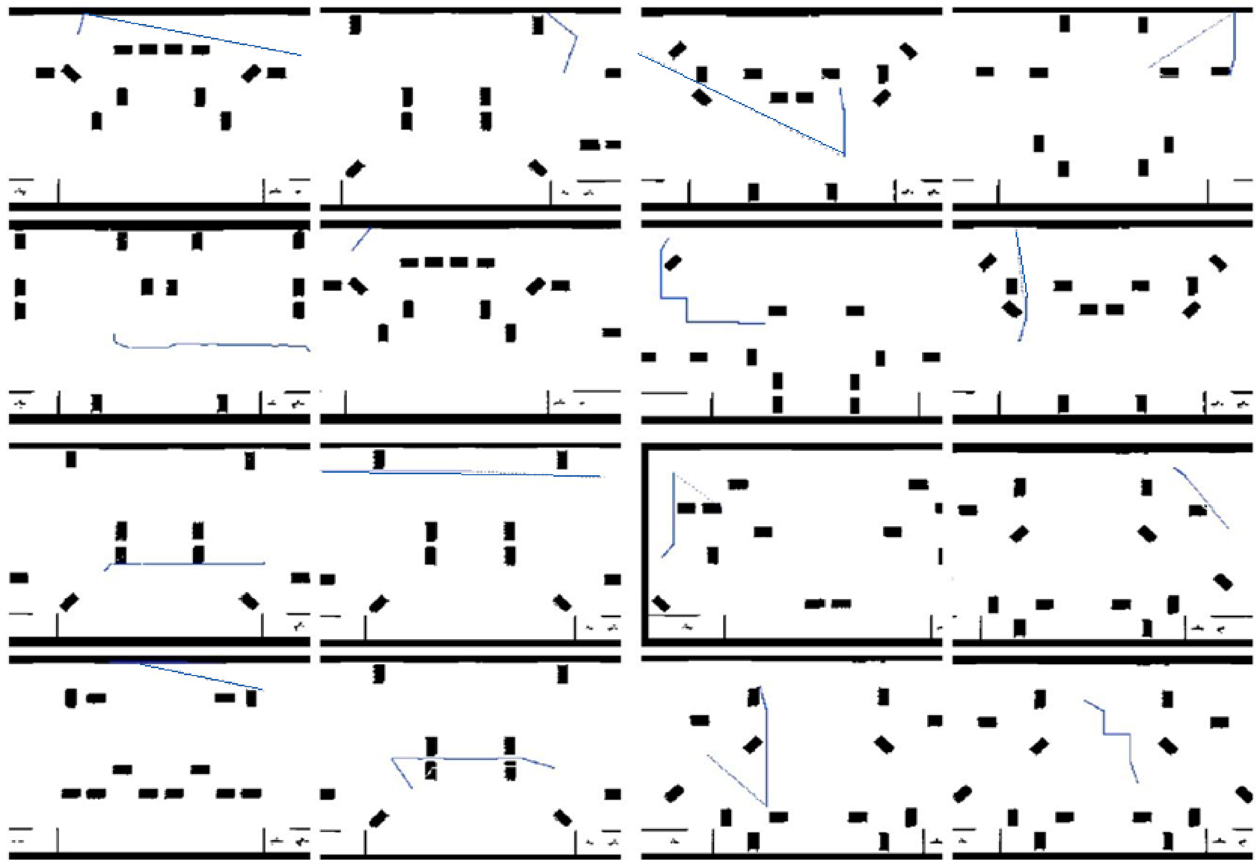


Figure 27 : Quatrième groupe de tester

5 Problème rencontré

Nous avons utilisé une méthode combinant deux réseaux de neurones pour prévoir la trajectoire. Les deux réseaux sont CNN et LSTM. CNN est utilisé pour identifier les emplacements des obstacles et LSTM est utilisé pour prévoir la vitesse. L'idée est que pendant la phase d' entraînement, le points de départ et la destination d'une trajectoire et toutes les points de vitesses(vitesse de l'abscisse, vitesse de l'ordonnée et vitesse angulaire) toutes les 0,1 secondes sont attribués aux réseaux de neurones en tant que données d' entraînement, puis les deux réseaux de neurones sont connectés pour les entraîner ensemble. Grâce à ce processus de formation, le robot peut apprendre à identifier l'emplacement de l'obstacle, tout en apprenant également à prévoir la vitesse. lors de la phase de prévision, nous fournissons la point de départ, la destinations et la cartes, afin que le réseau puisse utiliser ces conditions et calculer de l'etat suivant et continuer à se déplacer pas à pas vers la destination.

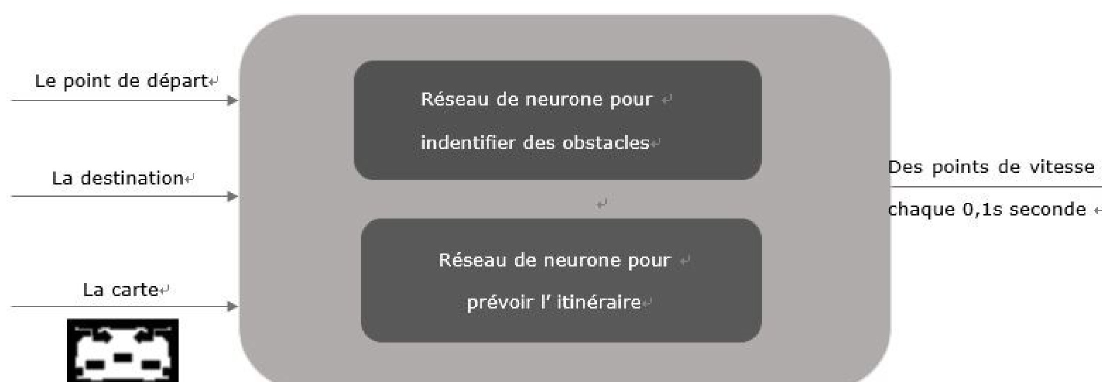


Figure 28 : Structure de combination des deux réseaux de neurones

Après avoir construit les deux réseaux, nous avons utilisé les 1800 trajectoires que nous avons collecté pour entraîner les réseaux, mais au stade de la prédiction de la vitesse, nous avons constaté que la vitesse prévu était très similaire à la vitesse réelle dans les 5 premières secondes, mais l'erreur devient très gros après 5 secondes. Nous avons ensuite essayé de remplacer LSTM par RNN et GRU, et modifié la méthode de connexion de deux réseaux afin de prévoir la vitesse, mais l'effet n'est toujours pas satisfaisant.

La figure ci-dessous représente les 25 trajectoires prévues, la curbe bleue correspondant à la vitesse de l'axe des x donnée par le simulateur et la curbe rouge correspondant à la vitesse prévue. Les images de vitesse de l'ordonnée et de vitesse angulaire sont similaires au image présentée ici.

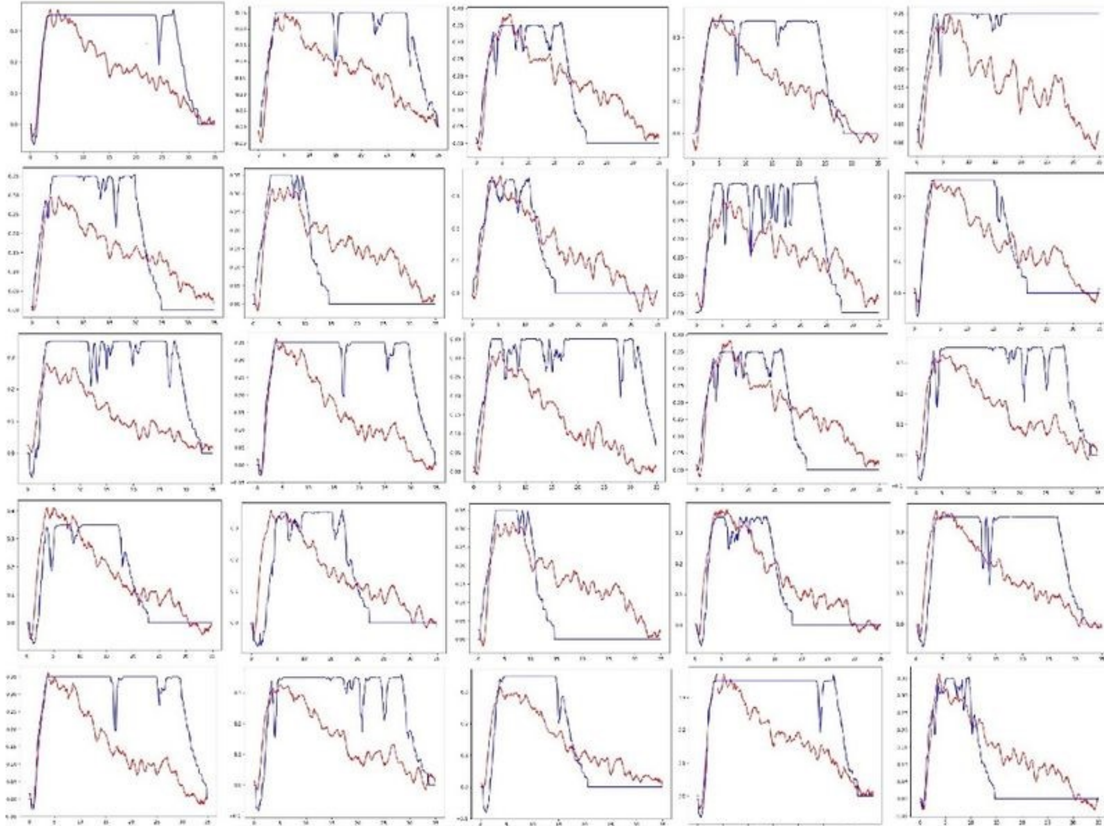


Figure 29 : Prédiction de la vitesse sous combinaison de deux réseaux de neurones

Nous pensons que la raison pour laquelle il peut très bien s'adapter au début est peut-être due au fait que, pour la vitesse, il doit y avoir une tendance à la hausse allant de 0 au maximum, et notre réseau ne peut que l'apprendre ça. Et puis l'erreur devient de plus en plus grande, nous pensons qu'il peut y avoir trois raisons:

1. Cela peut être dû au fait que la séquence de vitesse dont nous avons besoin est trop longue pour provoquer une explosion de gradient.

2. L'espace est trop grand et le nombre d'échantillons est trop petit, autrement dit, la carte est trop grande et les trajectoires utilisé pour former le réseau sont trop court.

3. Nous divisons la vitesse en vitesse angulaire, en vitesse X et en vitesse Y pour entraîner le réseau. Lorsque nous prédisons la vitesse, nous prévoyons également ces trois vitesses séparément, ce qui peut perturber la relation entre les trois vitesses elles-mêmes.

A la fin, après avoir consulter de nombreuses informations, nous avons décidé d'abandonner l'idée de connecter deux réseaux de neurones et de passer à l'apprentissage par renforcement DQN.

Conclusion

En conclusion, ce projet fût intéressant pour nous sur l'aspect de Machine Learning. Parce que l'intelligence artificielle est l'un des sujets à la mode aujourd'hui et ce projet nous permet de comprendre les principes de l'apprentissage automatique et son application pratique. Pendant le processus du projet, nous avons rencontré beaucoup de difficultés : choix d'entrées et de sorties, collection de données, conception de réseaux neuronaux et d'algorithmes, etc. C'est vraiment un défi pour nous.

Ce projet nous a permis de développer de nombreuses compétences techniques comme la compétence de recherches, la compétence de programmation en Python, la compétence de modélisation de réseau neuronal, etc. De plus, nous avons également appris plusieurs techniques nouvelles pendant le processus du projet. Nous avons compris la connaissance de Tensorflow, la connaissance de modèles différentes de Machine Learning et ses applications.

De plus, ce projet nous a permis de développer de nombreuses compétences humaines comme le travail en équipe, l'organisation de Wiki, etc.

Nous remercions à nos tuteurs, de nous proposer les méthodes très utiles pour résoudre les problèmes sur la construction de nos réseaux. Il nous a également donné l'opportunité de développer notre projet professionnel. Au long de ce projet, nous avons renforcé nos expériences de jeune ingénieur en devenir.