

Rapport sur le stage effectué du 03/06/19 au 05/09/19

dans l'Unité Mixte de Recherche de Sorbonne Université :

LIP6 (Laboratoire d'informatique de Paris 6)
Equipe MLIA (Machine Learning for Information Access)



Apprentissage statistique embarqué sur Raspberry Pi

Gorinskaia Katerina

Master 1 Ingénierie de la Robotique et des Systèmes Intelligents

À l'université
Sorbonne Université



Paris, 2019

Plan:

- Sommaire
- Présentation du laboratoire
- Travail du stage
 - Contexte
 - Exposé des travaux
 - Partie Robotique
 - Partie Simulation
 - Partie Traitement des images
 - Partie Machine Learning
- Conclusion
- Glossaire
- Annexe
 - Algorithmes
 - Liens et Documentations
- Code

Notation:

Couleur violet — voir glossaire.

Couleur bleu — voir annexe.

Sommaire

L'apprentissage statistique est un domaine de l'informatique centrée sur l'analyse et l'interprétation des modèles et des structures de données, ce qui permet aux agents intelligents d'apprendre, de raisonner et de prendre des décisions en dehors de l'interaction humaine. En même temps, l'adaptation de ces algorithmes à un usage en temps réel (par exemple, sur les systèmes embarqués) est encore freinée par les besoins computationnels très importants et nécessite généralement un support matériel de dernière génération.

Dans le cadre de mon Master d'Ingénierie de la robotique et des systèmes Intelligents, j'ai réalisé mon stage dans un laboratoire informatique **LIP6** afin de me familiariser avec les dernières techniques récentes en apprentissage statistique sur un support matériel embarqué.

L'objectif de mon stage était d'explorer les possibilités d'exploitation des techniques de Machine Learning sur la plateforme matérielle — un Raspberry Pi embarqué sur un robot véhiculé. Le but du projet était de développer une intelligence artificielle pour permettre au robot de suivre de manière intelligente la cible lui étant assignée. Ce travail permettait d'explorer l'adaptation de différents algorithmes d'apprentissage statistique, en particulier **l'analyse d'images** pour la reconnaissance de la cible, le **traitement de données des capteurs** pour l'orientation et la détection d'obstacles, et l'**apprentissage par renforcement** pour obtenir des politiques de jeu optimales.

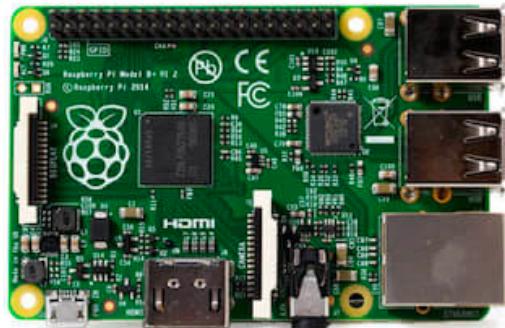
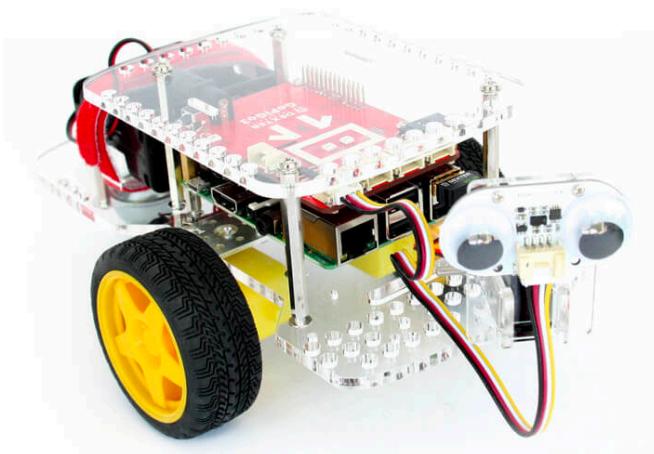


Photo: <https://www.dexterindustries.com/product/gopigo-beginner-starter-kit/>

Présentation du laboratoire

LIP6

Le LIP6, Unité Mixte de Recherche de Sorbonne Université et du Centre National de la Recherche Scientifique est un laboratoire de recherche en informatique. La recherche est réalisée au sein de vingt-deux équipes articulées autour de quatre axes transversaux:

- Intelligence artificielle et sciences des données (AID)
- Architecture, systèmes et réseaux (ASN)
- Sécurité, sûreté et fiabilité (SSR)
- Théorie et outils mathématiques pour l'informatique (TMC)

Dédié à la modélisation et à la résolution de problèmes fondamentaux liés aux applications, ainsi qu'à la mise en œuvre et à la validation par le biais de partenariats académiques et industriels, le LIP6 est le plus gros laboratoire de recherche en informatique en France.

CHIFFRES CLÉS

618 personnes au LIP6

215 membres permanents du personnel

- 160 enseignants chercheurs
- 26 chercheurs CNRS
- 6 chercheurs INRIA
- 10 membres du personnel administratif
- 13 ingénieurs

403 personnels non permanents dont :

- 180 doctorants
- 19 post-doctorants
- 16 ingénieurs en CDD
- 133 stagiaires

Actions d'excellence

- 4 ERC
- 2 chaires
- 3 LabEx
- 1 Equipex
- 4 membres de l'IUF

Publications 2014 - 2019

- 694 articles de revue
- 1644 conférences
- 261 thèses
- 37 HDR

Source: <https://www.lip6.fr/>

Équipe : MLIA - Machine Learning and Information Access

Le thème central du travail de l'équipe est l'apprentissage statistique, l'accent étant mis sur les aspects algorithmiques et l'application de l'analyse de données sémantiques appliquées. Les principaux objectifs de ces recherches sont les suivants:

- Apprentissage par représentation et apprentissage en profondeur:
- Structuration des résultats
- Apprentissage séquentiel et apprentissage par renforcement:

Domaines d'application:

Vision par ordinateur

Les chercheurs de l'équipe ont mis au point plusieurs modèles pour la détection et la reconnaissance de modèles visuels. Ils explorent toutes les approches basées sur la vision par ordinateur, la modélisation bio-inspirée et les stratégies d'apprentissage approfondi pendant de nombreuses années.

Traitemet du langage naturel et recherche d'information, recommandation

La MLIA est impliquée dans la communauté des textes depuis des années avec une contribution sur l'apprentissage du classement et sur la récupération de données semi-structurées. Les chercheurs ont développé des modèles pour la révision conjointe, la polarité du texte et la prédiction de recommandations.

Analyse dynamique complexe des données

Les chercheurs de la MLIA ont établi des modèles statistiques pour l'analyse et la modélisation de la diffusion d'informations sur les réseaux sociaux en formulant le problème dans des espaces continus plutôt que dans les espaces discrets. Ils ont également développé une coopération avec des acteurs tels que Renault et STIF pour analyser les problèmes dans le domaine des transports.

L'équipe participe à plusieurs collaborations académiques internationales et nationales et a développé une coopération étroite avec des partenaires de R&D industriels, à la fois par la participation conjointe à des projets et par des contrats bilatéraux.

La MLIA est impliquée dans le service communautaire international et national. Au niveau local, l'Université Sorbonne participe activement à la fois à la recherche et à l'enseignement.

Travail du stage

Contexte

L'objectif de ce stage était d'explorer les possibilités et les applications d'un processeur **Raspberry Pi** dans le cadre d'un système embarqué. Il abordait les thèmes de la robotique, de la programmation et du traitement d'images.

Tout au long de mon stage, je devais travailler avec un processeur **Raspberry Pi** sous un système d'exploitation **Raspbian** (basé sur Debian) et créer une simulation 3D à l'aide d'une bibliothèque **Python Panda3D** afin d'effectuer des tâches d'apprentissage. J'ai également dû utiliser la bibliothèque **OpenCV** pour gérer le traitement des images à la fois en simulation et sur un vrai robot.

Exposé des travaux

Partie Robotique

Mon travail a commencé par les études du microprocesseur Raspberry Pi. Je travaillais avec **Raspberry Pi 3 Model B**, en utilisant un système d'exploitation spécifique, Raspbian. J'accédais à la ligne de commande du Pi depuis mon ordinateur via **SSH**, et le **VNC** permettait l'accès à distance à l'interface graphique.

J'ai ensuite dû étudier la documentation technique d'une plate-forme robotique, **GoPiGo de Dexter Industries**, dont les capteurs les plus remarquables et les plus utilisés étaient une caméra et un capteur de distance laser.

La classe principale que j'ai créée, **RobotDexter**, décrit un modèle physique de robot. Elle comprend l'initialisation d'une caméra, un capteur de distance et des fonctions de commandes liées aux opérations physiques exécutées par un robot: *set_speed*, *get_offset*, *shutdown*, *reset*, *odometry*, *get_image* et *get_dist*.

set_speed est directement lié aux servomoteurs d'un robot et permet à celui-ci d'effectuer un mouvement vers l'avant, ainsi que des manœuvres en virage.

get_offset renvoie les valeurs de deux **codeurs rotatifs**, intégrés aux roues.

get_image capture un cadre d'image et le transmet à un algorithme de traitement des images.

get_dist renvoie la valeur de distance d'un capteur laser.

odometry utilise les valeurs obtenues par *get_offset* pour effectuer les calculs, permettant ainsi l'alignement des deux roues.

shutdown mets la vitesse d'un robot à 0.

reset aligne les deux roues pour assurer un mouvement en ligne droite.

La commande d'un robot est effectuée par une série des classes contrôleurs, décrit dans le fichier **Controller.py**. Chaque contrôleur a la même structure: méthodes d'initialisation, de démarrage, d'arrêt et de mise à jour. La méthode de mise à jour s'est avérée nécessaire pour que le robot puisse répondre à l'environnement en temps réel.

Dans un premier temps, j'ai dû mettre en œuvre des commandes de base pour se déplacer en ligne droite et pouvoir [tourner à un angle donné](#).

Difficultés rencontrées:

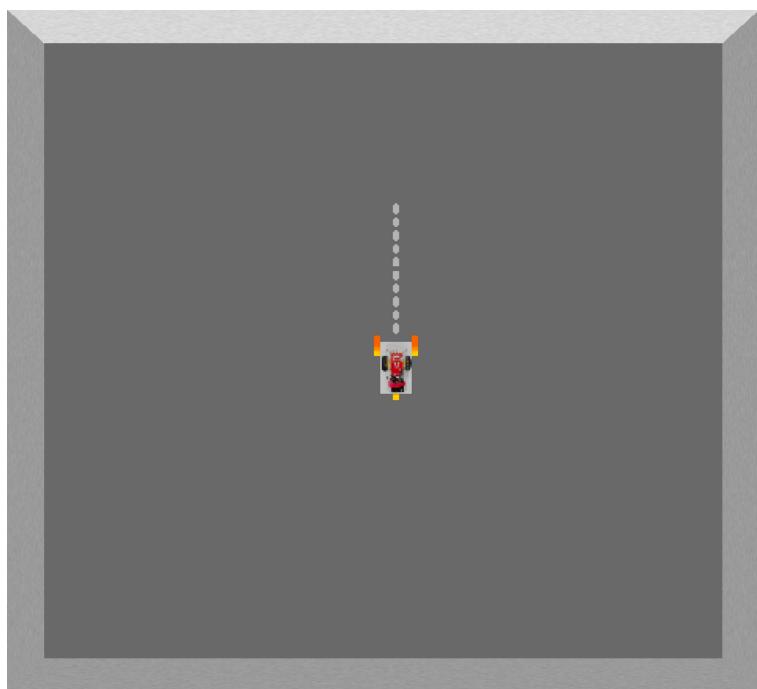
Connexion à distance. Se connecter à Raspberry Pi à distance s'est révélé un défi en raison de la spécification du réseau Wi-Fi en laboratoire.

Structure du code vs Puissance de Raspberry Pi. Un problème récurrent tout au long de mon stage: j'ai toujours dû prendre en compte les limites techniques de cette plate-forme, et plus précisément, limiter le temps de calcul dans la méthode de mise à jour pour garantir le comportement en temps réel.

Partie Simulation

Dans cette partie, je devais construire un modèle de l'environnement et un robot dans un espace 3D, et programmer son comportement pour qu'il soit aussi proche que possible du monde réel.

Pour cette tâche, j'ai utilisé la bibliothèque Python [Panda3D](#) et [Bullet Physics Library](#) - un moteur physique simulant la détection de collision et la dynamique des objets.



Capture d'écran lors d'une simulation.

Le modèle possédait 3 roues, comme le vrai robot: deux roues avant avec moteurs attachés et une troisième roue arrière pour l'équilibre.

La commande du robot réel et du modèle 3D devait provenir d'un contrôleur unique. Pour cela, une classe du modèle 3D devait avoir des fonctions physiques identiques au vrai robot afin d'assurer un comportement similaire. L'imitation de ces capteurs sera discutée plus loin.

Difficultés rencontrées:

Codeur rotatif. L'une des différences les plus notables entre les deux robots était la présence d'un codeur rotatif sur des roues - non seulement ce dernier était nécessaire afin d'assurer le mouvement en avant, mais ce dernier était également crucial de calculer l'angle en cas de virage.

Dans le cadre de la simulation 3D, [j'ai modélisé les résultats d'un codeur](#) en utilisant le positionnement des roues par rapport à l'environnement.

Capteur de distance. Comme la modélisation d'un faisceau laser était impossible en raison des limites de la simulation, j'ai utilisé plusieurs détecteurs de collision pour construire une approximation du capteur de distance. Il était représenté par une série de sphères, chacune affectée à une valeur de distance différente.

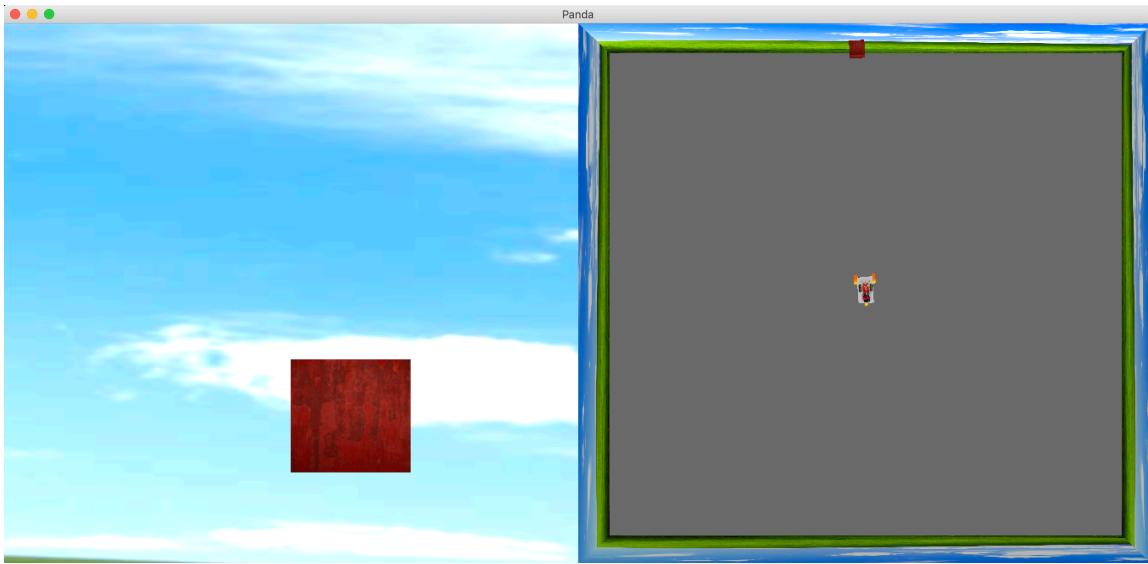
Mêmes valeurs de vitesse et de distance. Pour que la même commande puisse être comprise de la même manière par le robot et son modèle 3D, les valeurs de vitesse et de distance de collision doivent être ajustées de la même manière. Cela comprenait l'ajout d'une force de freinage supplémentaire sur les roues d'un robot simulé, la modélisation d'une certaine manière des frictions avec le sol.

Partie Traitement des images

L'objectif principal de la tâche étant de pouvoir suivre une cible spécifique, j'ai décidé de mettre en œuvre l'algorithme permettant de suivre un objet rouge dans l'espace.

Pour tester l'algorithme en pratique, j'ai utilisé la classe **ControllerFollow**, qui prenait les coordonnées de centroïde d'une cible, les comparait aux valeurs correspondant au centre de l'image, et ajustait les valeurs transmises aux moteurs de roue. Si le centroïde se trouvait dans la partie gauche de l'écran, le robot tournait à gauche, et vice-versa. Les photos étaient prises dans un thread différent, pour ne pas bloquer les autres fonctionnalités du robot.

Dans le cas d'un modèle 3D, j'ai utilisé des captures d'écran d'une deuxième caméra montée sur le modèle.



Capture d'écran lors d'une simulation.

Difficultés rencontrées:

Puissance de Raspberry Pi. Lors des simulations, le robot s'acquittait parfaitement de sa tâche, alors que il fallait près de 1,5 seconde pour effectuer le même traitement d'une image sur un Pi. C'est un résultat absolument naturel et prévu, soulignant à nouveau l'importance d'avoir un modèle 3D pour pouvoir tester le comportement d'un robot sans les retards.

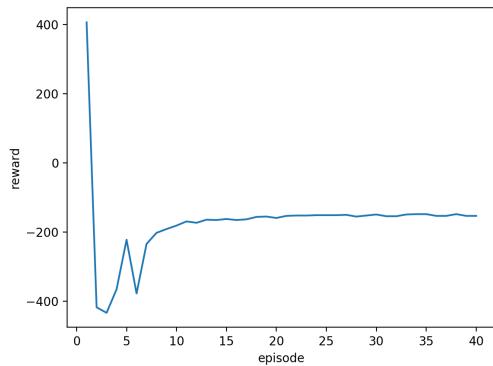
Partie Machine Learning

Au cours de cette étape, j'ai mis en œuvre 2 algorithmes d'apprentissage: **Q-learning** et **réseau de neurones (perceptron multicouche)**.

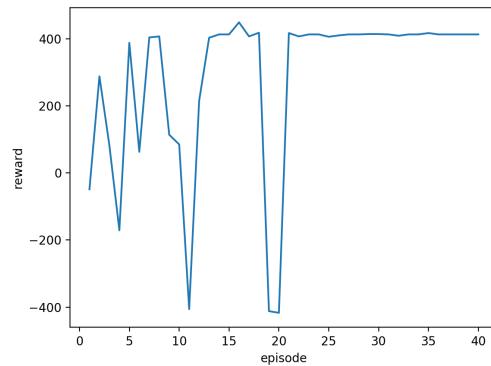
La tâche de test la plus simple pour implémenter l'algorithme **Q-learning** dans un contexte de robot était de se rapprocher le plus possible du mur sans le toucher.

En utilisant des valeurs de distance discrètes en tant qu'états et des valeurs de vitesse en tant qu'actions, j'ai mis en place un système simple de récompenses. En quelques épisodes, le robot a réussi à trouver une politique optimale. Sur les figures ci-dessous, l'on peut voir l'évolution de la récompense cumulée sur plusieurs épisodes.

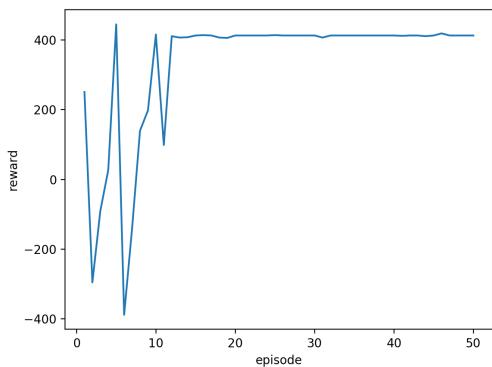
La figure suivante présente la même évolution avec différentes valeurs d'un paramètre gamma (discount rate, facteur d'actualisation).



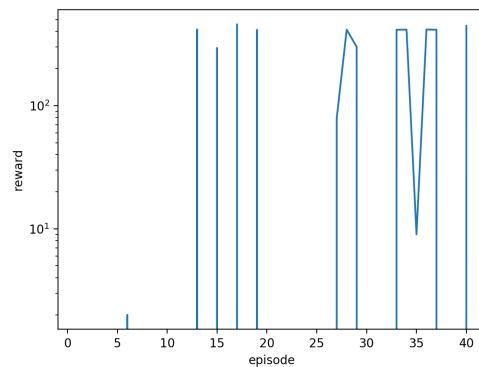
$$\gamma = 0$$



$$\gamma = 0.1$$



$$\gamma = 0.4$$



$$\gamma = 0.9$$

Le **facteur d'actualisation gamma** détermine l'importance des récompenses futures. Un facteur de 0 rendra l'agent "myope" en ne tenant compte que des récompenses actuelles, alors qu'un facteur proche de 1 le poussera à rechercher une récompense élevée à long terme. Si le facteur de réduction atteint ou dépasse 1, les valeurs d'action peuvent diverger.

À la fin, avec un gamma = 0,4, le robot a appris une politique optimale après environ 15-20 épisodes.

Deuxième algorithme que j'ai implémenté était le **perceptron multicouche (Multilayered Perceptron, MLP)**, ou le réseau de neurones artificiels.

Les MLP avec une couche cachée sont capables d'approximer n'importe quelle fonction continue, ce qui correspond exactement à la tâche à accomplir. Au lieu d'utiliser les états et les récompenses discrets, un réseau de neurones s'est entraîné sur quelques entrées et sorties pour pouvoir calculer les valeurs de vitesses en fonction des données des capteurs.

J'ai choisi d'utiliser le réseau des neurones avec une couche cachée avec quatre neurones à l'intérieur. Il est important de noter que la formation a été effectuée sur une simulation - la puissance de Pi n'est pas suffisante pour de telles tâches.

Les poids finaux des neurones d'un modèle de travail ont été enregistrés et transférés à la classe **ControllerForwardSmart**, un contrôleur chargé d'utiliser ce réseau de neurones pour effectuer l'une des deux tâches suivantes: **se rapprocher le plus possible du mur ou suivre une cible.**

Difficultés rencontrées:

Réglage des hyperparamètres. Afin d'assurer des résultats optimales, il était nécessaire d'effectuer une série de tests et de régler manuellement les paramètres de mes modèles.

Resultat

Comme la simulation et le robot fonctionnent avec les mêmes commandes, il restait à ajouter des contrôleurs, chargés d'exécuter le comportement requis, et de les mettre dans l'entrée d'un contrôleur de séquence. Après le lancement du script, les deux options sont présentées à l'utilisateur qui peut choisir l'une ou l'autre.

Le script est ensuite exécuté et il est facile de tester différentes valeurs de vitesse et de distance de collision.

Grâce au réseau de neurones, mon robot a pu apprendre l'algorithme permettant de suivre une cible choisie. En raison des limitations de Raspberry Pi, ses performances n'étaient pas optimales, mais grâce à la simulation, j'ai pu tester l'algorithme et m'assurer qu'il fonctionnait correctement.

Conclusion

Cet algorithme peut ultérieurement être mis en œuvre sur des plates-formes embarquées dans le cadre de multiples problèmes pratiques. Des cas similaires incluent une valise suivant son propriétaire, des robots de ménage ou des robots dans un centre de tri.

Je pense que ce stage a été très fructueux. J'ai appris énormément de choses et j'ai pu utiliser mes connaissances antérieures en robotique, intelligence artificielle, systèmes embarqués et reconnaissance d'images sur un cas pratique. Ce stage a suscité mon intérêt pour l'apprentissage statistique et mes recherches personnelles m'ont permis d'approfondir ma compréhension de ce sujet particulier.

Dans le laboratoire qui m'a accueillie pendant deux mois, il y a une très bonne ambiance, et les recherches menées présentent une valeur indéniable. J'ai été très fière d'avoir pu participer, même brièvement, à ce travail.

Forte de cette expérience, j'aimerais beaucoup par la suite essayer de m'orienter via un prochain stage vers le secteur des systèmes embarqués et, plus précisément, Machine Learning at Traitement des images.

Ce domaine passe par une période de développement rapide, avec les nouveaux algorithmes optimisés développés chaque jour et je suis sûre de pouvoir contribuer activement à ces recherches.

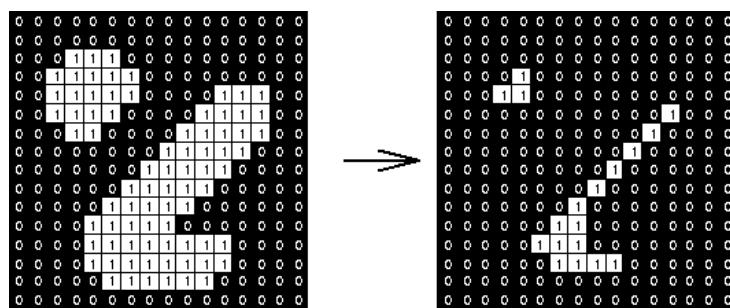
Glossaire

SSH (Secure Shell) — un protocole de réseau cryptographique qui fournit une authentification forte et des communications de données cryptées entre deux ordinateurs se connectant via un réseau ouvert tel qu'Internet

VNC (Virtual Network Computing) — un système graphique de partage d'écran afin de contrôler à distance un autre ordinateur.

Codeur rotatif — un dispositif électromécanique qui convertit la position angulaire ou le mouvement d'un axe en signaux de sortie analogiques ou numériques.

Erosion — l'une des deux opérations fondamentales (l'autre étant la dilatation) dans le traitement d'images morphologiques. L'érosion d'une image binaire f par un élément structurant s (noté $f \ominus s$) produit une nouvelle image binaire $g = f \ominus s$ avec des 1 en tous les emplacements (x, y) de l'origine d'un élément structurant auquel cet élément structurant s correspond à l'image d'entrée f . C'est-à-dire que $g(x, y) = 1$ correspond à s et f et à 0 sinon, pour toutes les coordonnées de pixel (x, y) .



Source: <https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm>

HSV (hue, saturation, value) — une représentation alternative du modèle de couleur RGB. Dans ce modèle, les couleurs de chaque teinte sont disposées en une tranche radiale, autour d'un axe central de couleurs neutres allant du noir en bas au blanc en haut. La représentation HSV modélise la façon dont les peintures de différentes couleurs se mélangent, la dimension de saturation ressemblant à différentes teintes de peinture de couleur vive et la dimension de valeur ressemblant au mélange de ces peintures avec des quantités variables de peinture noire ou blanche.

Q-learning — un algorithme d'apprentissage par renforcement sans modèle. Le but de Q-learning est d'apprendre une politique avec un agent, de prendre des décisions en fonction des circonstances. Il ne possède pas de modèle d'environnement et peut résoudre les problèmes de transitions et de récompenses stochastiques sans nécessiter d'adaptations. L'apprentissage par renforcement implique un agent, un ensemble d'états S et un ensemble A d'actions par état. En effectuant une action a in A , l'agent passe d'un état à l'autre. L'exécution d'une action dans un état spécifique fournit à l'agent une récompense (un score numérique).

L'algorithme a une fonction qui calcule la qualité d'une combinaison état-action Q pour chaque pas (équation de Bellman):

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

Un perceptron multicouche (Multilayered Perceptron, MLP) — un réseau neuronal artificiel profond. Il est composé d'un unique classifieur linéaire: une couche d'entrée pour recevoir le signal, une couche de sortie qui prend une décision ou une prédiction sur l'entrée, et un nombre arbitraire de couches masquées qui constituent le véritable moteur de calcul du processeur MLP.

Annexe

Algorithmes

Codeurs pour le Model 3D.

Pour obtenir le décalage par rapport aux roues, j'ai utilisé les coordonnées absolues de chaque nœud d'une roue:

```
self.curr_posl = self.wheelL.getPos()
self.curr_posr = self.wheelR.getPos()
x2l, y2l = (self.curr_posl[0], self.curr_posl[1])
x2r, y2r = (self.curr_posr[0], self.curr_posr[1])
```

La distance totale, couverte par les roues, était réinitialisée à chaque *reset* et je l'augmentais progressivement en ajoutant une différence carrée entre la dernière position de la roue et sa position actuelle:

```
self.total_distl += math.sqrt(pow((x2l - self.x1l), 2) + pow((y2l - self.y1l), 2))
self.total_distr += math.sqrt(pow((x2r - self.x1r), 2) + pow((y2r - self.y1r), 2))
```

La position actuelle a ensuite été mise à jour pour devenir la dernière position et la distance totale, correspondant à la valeur d'un codeur incrémental rotatif, a été renvoyée en tant que sortie:

```
self.x1l, self.y1l = (x2l, y2l)
self.x1r, self.y1r = (x2r, y2r)

res = (self.total_distl, self.total_distr)
return res
```

Odométrie.

L'odométrie est l'utilisation des données des capteurs de mouvement pour estimer le changement de position au fil du temps. Dans mon cas, je l'ai utilisée pour m'assurer que les deux roues avancent à la même vitesse - la principale raison étant l'imperfection possible d'un véhicule robotisé.

Tout au long de ce projet, j'ai utilisé plusieurs constantes, qui définissaient les propriétés des roues:

WHEEL_BASE_WIDTH — distance (mm) de la roue gauche à la roue droite.
WHEEL_DIAMETER — diamètre de la roue (mm)

```

WHEEL_BASE_CIRCUMFERENCE = WHEEL_BASE_WIDTH * math.pi - périmètre du cercle de
rotation (mm)
WHEEL_CIRCUMFERENCE = WHEEL_DIAMETER * math.pi - périmètre de la roue (mm)

```

L'algorithme consistait à ajouter une petite valeur (facteur de calibration) aux deux roues afin d'augmenter sa vitesse si nécessaire.

Si N est un nombre de pas, D - diamètre de la roue, alors la distance par pas peut être trouvée comme suit:

$$\text{Distance / Pas} = (\text{Pi} * \text{D}) / \text{N} = \text{C, facteur de calibration}$$

```

left_steps, right_steps = self.get_offset()
if left_steps>0 and right_steps>0:
    cl = self.WHEEL_CIRCUMFERENCE / left_steps
    cr = self.WHEEL_CIRCUMFERENCE / right_steps

```

Après avoir trouvé les deux facteurs pour les deux roues, j'utilise leur différence pour influencer l'un des deux coefficients (roue gauche ou droite):

```

coeff = abs(cl-cr)
if cr<cl:
    cl = 1
    cr = 1+coeff
else:
    cr = 1
    cl = 1+coeff
return cl, cr

```

Déplacement angulaire.

Cette fonction utilise les valeurs de décalage des roues et calcule une rotation d'angle à chaque étape. La condition d'arrêt est le fait d'atteindre l'angle donné.

Si D est un diamètre de la roue, p - le périmètre du cercle de rotation, d - le décalage de la roue (encodeur rotatif), l'angle sera alors calculé comme suit:

$$\text{Angle} = ((\text{D} * \text{d}) / \text{p}) / 2$$

```

res = self.robot.get_offset()
offset = max(abs(res[1]), abs(res[0]))
turn = ((self.robot.WHEEL_CIRCUMFERENCE*offset)/(self.robot.WHEEL_BASE_CIRCUM-
FERENCE))/2

```

La sortie de la fonction est une condition d'arrêt:

```
return abs(turn)>=abs(self.angle)
```

Traitement d'images.

Le prétraitement consistait en une simple égalisation d'histogramme.

```
def equalize_hist(self, img):
    img_yuv = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)
    img_yuv[:, :, 0] = cv2.equalizeHist(img_yuv[:, :, 0])
    img_output = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)
    return img
```

À l'aide d'un masque binaire, j'ai séparé la région des variations du couleur rouge (en HSV) puis effectué une **érosion** pour éliminer le bruit de point:

```
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

mask1 = cv2.inRange(hsv, (0,50,20), (5,255,255))
mask2 = cv2.inRange(hsv, (175,50,20), (180,255,255))
mask = cv2.bitwise_or(mask1, mask2)

kernel = np.ones((5,5),np.uint8)
mask = cv2.erode(mask,kernel,iterations = 1)
```

La prochaine étape consistait à trouver une région correspondant à ma cible. Pour cela, j'ai vérifié l'ensemble des régions, en éliminant en même temps les plus petites pour assurer un calcul plus rapide, et parmi le reste, j'ai choisi la plus grande cible. La valeur moyenne de tous les pixels correspondants était la coordonnée d'un centroïde.

Dans le cas d'une photo vide ou d'une image sans cible, les coordonnées ont été assignées au centre de l'écran. CAMX et CAMY désignaient les valeurs x et y de la résolution de l'image.

```
(cnts, _) = cv2.findContours(mask.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)

try:
    for cnt in cnts:
        if cv2.contourArea(cnt)<CAMX/5: # Don't count the noise
            pass
        else:
            c = max(cnts, key = cv2.contourArea)
            res = np.mean(c, axis=0)
            res = res[0]
            self.cx = int(round(res[0]))
            self.cy = int(round(res[1]))
except : # Empty photo
    self.cx = CAMX/2
    self.cy = CAMY/2
```

Ces coordonnées ont été transmises en tant que sortie au contrôleur.

On peut voir la progression d'algorithme sur les images ci-dessus.



De haut à gauche en bas à droite: image initiale, recherche des régions d'intérêt, image binaire après érosion, coordonnées finales.

Perceptron multicouche.

Tout d'abord, il est nécessaire d'initialiser le réseau de neurones et toutes ses couches.

```
layer1 = NeuronLayer(4, 1) # 4 neurons, 1 input
layer2 = NeuronLayer(1, 4) # 1 neuron, 4 inputs (output)
self.neural_network = NeuralNetwork(layer1, layer2)
```

L'initialisation consiste à attribuer des poids aléatoires à chaque nœud:

```
self.weights = 2 * random.random((number_of_inputs_per_neuron, number_of_neurons)) - 1
```

L'ensemble de données d'entraînement des valeurs d'entrée pour la tâche de distance consiste en des distances entre le robot et le mur. Il est ensuite normalisé pour être compris entre 0 et 1.

```
training_set_inputs = array([[15.0],[30.0], [60.0], [75.0], [90.0], [150.0],
[300], [1000.0]])
training_set_inputs = self.normalize(training_set_inputs)
training_set_outputs = array([[ 0, 0.1, 0.2, 0.5, 0.7, 0.95, 0.99, 1]]).T
```

Le processus d'entraînement commence.

```
self.neural_network.train(training_set_inputs, training_set_outputs, 50000)
```

Je passe la somme pondérée des entrées à travers la fonction sigmoïde pour la normaliser entre 0 et 1.

Je forme ensuite le réseau de neurones à travers un processus d'essais et d'erreur en ajustant les poids à chaque fois. Pour chaque itération, je transmets l'ensemble de formation via le réseau de neurones:

```
output_layer_1, output_layer_2 = self.forward(training_set_inputs)
```

Je calcule ensuite l'erreur pour la deuxième couche en faisant la différence entre le résultat souhaité et la sortie prévue:

```
layer2_error = training_set_outputs - output_layer_2
layer2_delta = layer2_error * self.__sigmoid_derivative(output_layer_2),
```

Lorsque la fonction dérivée sigmoïde représente le gradient de la courbe sigmoïde, elle indique une confiance quant au poids existant.

Je fais les mêmes calculs pour la première couche.

Ensuite, je calcule l'ajustement pour les poids.

```
layer1_adjustment = training_set_inputs.T.dot(layer1_delta)
layer2_adjustment = output_layer_1.T.dot(layer2_delta)

self.layer1.weights += layer1_adjustment
self.layer2.weights += layer2_adjustment
```

À la fin de toutes les itérations, les poids sont stockés et peuvent être utilisés dans une simulation 3D ou un robot réel.

Chaque fois qu'une nouvelle valeur de distance arrive à une méthode de mise à jour, je la passe à travers le réseau de neurones et j'obtiens la vitesse qui en résulte.

```
new_input = (value - 15.0)/(1000.0 - 15.0)
hidden_state, output = self.neural_network.forward(array([new_input]))
return (output[0])
```

Liens et Documentations

- LIP6: <https://www.lip6.fr/>
- MLIA: <https://mlia.lip6.fr/>
- Raspberry Pi 3 Model B Datasheet: https://www.terraelectronica.ru/pdf/show?pdf_file=%252Fd%252Fpdf%252FT%252FTechicRP3.pdf
- Dexter Industries GoPiGo3 Documentation: <https://gopigo3.readthedocs.io/en/master/>
- Panda3D Documentation: <https://www.panda3d.org/manual/>
- OpenCV Documentation: <https://docs.opencv.org/4.1.0/>
- Distance sensor Datasheet: <https://www.st.com/content/ccc/resource/technical/document/datasheet/group3/b2/1e/33/77/c6/92/47/6b/DM00279086/files/DM00279086.pdf/jcr:content/translations/en.DM00279086.pdf>

Code

RobotDexter.py

```
import time
import math
from images import Image_Processing
from picamera import PiCamera

class Dexter:
    'Robot class'
    WHEEL_BASE_WIDTH      = 117 # distance (mm) de la roue gauche a la roue
droite.
    WHEEL_DIAMETER        = 66.5 # diametre de la roue (mm)
    WHEEL_BASE_CIRCUMFERENCE = WHEEL_BASE_WIDTH * math.pi # perimetre du cercle
de rotation (mm)
    WHEEL_CIRCUMFERENCE   = WHEEL_DIAMETER * math.pi # perimetre de la roue
(mm)
    CAMX = 320
    CAMY = 280

    def __init__(self, gpg):
        self.gpg = gpg
        self.dist_mm = gpg.init_distance_sensor()
        self.camera = PiCamera()
        self.camera.resolution = (self.CAMX, self.CAMY)
        self.camera.framerate = 30

    def set_speed(self, left_speed, right_speed):
        self.gpg.set_motor_dps(self.gpg.MOTOR_LEFT, left_speed)
        self.gpg.set_motor_dps(self.gpg.MOTOR_RIGHT, right_speed)

    def shutdown(self):
        self.set_speed(0,0)

    def reset(self):
        left_target, right_target = self.get_offset()
        self.gpg.offset_motor_encoder(self.gpg.MOTOR_LEFT, left_target)
        self.gpg.offset_motor_encoder(self.gpg.MOTOR_RIGHT, right_target)
        self.N = 0

    def get_offset(self):
        left_pos = self.gpg.get_motor_encoder(self.gpg.MOTOR_LEFT)
        right_pos = self.gpg.get_motor_encoder(self.gpg.MOTOR_RIGHT)
        return (left_pos, right_pos)

    def get_dist(self):
        return self.dist_mm.read_mm()

    def get_speed(self):
        return self.gpg.get_speed()
```

```

def condition(self, ctrl):
    return self.get_dist() <= ctrl.dist

def odometry(self):
    cl = 1
    cr = 1
    coeff = 0

    left_steps, right_steps = self.get_offset()
    if left_steps>0 and right_steps>0:
        cl = self.WHEEL_CIRCUMFERENCE / left_steps
        cr = self.WHEEL_CIRCUMFERENCE / right_steps
        coeff = abs(cl-cr)
        if cr<cl:
            cl = 1
            cr = 1+coeff
        else:
            cr = 1
            cl = 1+coeff

    return cl, cr

def get_image(self):
    file_name = 'result.jpg'
    self.camera.capture(file_name)
    img = Image_Processing(file_name, self.CAMX, self.CAMY)
    return img.coord()

```

RobotModel3D.py

```

from panda3d.bullet import *
from panda3d.core import *
from direct.task import Task

import time
import math

#--- Main class ---

class Robot (BulletVehicle):
    def __init__ (self, render, world, sim):
        self.sim = sim
        self.world = world
        WHEEL_BASE_WIDTH          = 0.0065
        WHEEL_DIAMETER             = 0.66
        self.WHEEL_BASE_CIRCUMFERENCE = WHEEL_BASE_WIDTH * math.pi
        self.WHEEL_CIRCUMFERENCE     = WHEEL_DIAMETER * math.pi
        self.CAMX = 320
        self.CAMY = 240

        self.total_distl = 0
        self.total_distr = 0
        self.last_posl = 0
        self.last_posr = 0
        self.xll, self.yll = (0, 0)
        self.xlr, self.ylr = (0, 0)

        # Chassis body
        shape = BulletBoxShape(Vec3(0.5,0.8,0.5))
        ts = TransformState.makePos(Point3(0, 0, 0.06))

        self.chassisNP = render.attachNewNode(BulletRigidBodyNode('Vehicle'))
        self.chassisNP.node().addShape(shape, ts)
        self.chassisNP.setPos(0, 0, 0)
        self.chassisNP.node().setMass(1)
        self.chassisNP.node().setDeactivationEnabled(False)
        self.chassisNP.setScale (0.5,0.9,0.5)

        self.robotModel = loader.loadModel("cube")      # Static Robot model
        self.robotModel.reparentTo (self.chassisNP) # Reparent the model to the
node

```

```

robot_tex = loader.loadTexture("textures/robot.jpeg")
self.robotModel.setTexture(robot_tex, 1)
self.world.attachRigidBody(self.chassisNP.node())

# Vehicle
super(Robot, self).__init__(world, self.chassisNP.node())
self.setCoordinateSystem(ZUp)
world.attachVehicle(self)

# Wheel Right
self.wheelR = loader.loadModel('models/wheel.egg')
self.wheelR.reparentTo(render)
self.addWheel(Point3(0.60, 0.75, 0.3), self.wheelR)

# Wheel Left
self.wheelL = loader.loadModel('models/wheel.egg')
self.wheelL.reparentTo(render)
self.addWheel(Point3(-0.60, 0.75, 0.3), self.wheelL)

# Wheel back
self.wheelB = loader.loadModel('models/wheel.egg')
self.wheelB.reparentTo(render)
self.addWheel(Point3(0, -0.75, 0.3), self.wheelB)

self.count = 1

def addWheel(self, pos, np):
    wheel = self.createWheel()
    wheel.setNode(np.node())
    wheel.setChassisConnectionPointCs(pos)
    wheel.setFrontWheel(True)

    wheel.setWheelDirectionCs(Vec3(0, 0, -1))
    wheel.setWheelAxeCs(Vec3(1, 0, 0))
    wheel.setWheelRadius(0.33)
    wheel.setMaxSuspensionTravelCm(40.0)

    wheel.setSuspensionStiffness(40.0)
    wheel.setWheelsDampingRelaxation(2.3)
    wheel.setWheelsDampingCompression(4.4)
    wheel.setFrictionSlip(100.0)
    wheel.setRollInfluence(0.1)

def set_speed(self, left_speed, right_speed):
    if left_speed == 0 and right_speed == 0:
        self.setBrake(100, 2)
        self.setBrake(50, 0)
        self.setBrake(50, 1)
        self.applyEngineForce(0, 0)
        self.applyEngineForce(0, 1)
    else:
        self.setBrake(0.23, 2)
        self.applyEngineForce(left_speed/20, 1)
        self.applyEngineForce(right_speed/20, 0)

def shutdown(self):
    self.set_speed(0,0)

def reset(self):
    self.sim.distance = 1000
    self.set_speed(0,0)
    self.total_distl = 0
    self.total_distr = 0

    self.last_posl = self.wheelL.getPos()
    self.last_posr = self.wheelR.getPos()
    self.xll, self.yll = (self.last_posl[0], self.last_posl[1])
    self.xlr, self.ylr = (self.last_posr[0], self.last_posr[1])

def get_offset(self):
    self.curr_posl = self.wheelL.getPos()
    self.curr_posr = self.wheelR.getPos()
    x2l, y2l = (self.curr_posl[0], self.curr_posl[1])

```

```

        x2r, y2r = (self.curr_posr[0], self.curr_posr[1])
        self.total_distl += math.sqrt(pow((x2l- self.x1l), 2) + pow((y2l- self.y1l), 2))
        self.total_distr += math.sqrt(pow((x2r- self.xlr), 2) + pow((y2r- self.ylr), 2))

        self.x1l, self.y1l = (x2l, y2l)
        self.xlr, self.ylr = (x2r, y2r)
        res = (self.total_distl, self.total_distr)
        return res

    def get_dist(self):
        return self.sim.distance

    def get_speed(self):
        return self.current_speed_km_hour

    def condition(self, ctrl):
        if ctrl.flag == False:
            self.sim.distance = 1000
        return ctrl.flag # Collision detections

    def odometry(self):
        cl = 1
        cr = 1
        coeff = 0

        left_steps, right_steps = self.get_offset()

        if left_steps>0 and right_steps>0:
            cl = self.WHEEL_CIRCUMFERENCE / left_steps
            cr = self.WHEEL_CIRCUMFERENCE / right_steps
            coeff = abs(cl-cr)
            if cr<cl:
                cl = 1
                cr = 1+coeff
            else:
                cr = 1
                cl = 1+coeff
        if cl>1 or cr>1:
            cl = 1
            cr = 1
        return cl, cr

    def get_image(self):
        return self.sim.take_screenshot(self.CAMX, self.CAMY)

```

main_simulation3d.py

```

from RobotModel3D import Robot

import sys
import math
import datetime
from images import Image_Processing

from direct.showbase.ShowBase import ShowBase
from panda3d.core import *
from panda3d.bullet import *
from direct.task import Task
from direct.showbase.InputStateGlobal import inputState

class Simulation(ShowBase):

    def __init__(self):
        import direct.directbase.DirectStart

        base.cTrav = CollisionTraverser()
        #base.disableMouse()

        self.collHandEvent = CollisionHandlerEvent()

```

```

    self.collHandEvent.addInPattern('into-%in')

    self.robot = Robot(worldNP, world, self)
    self.sColl = self.initCollisionSphere(self.robot.robotModel, True,
Point3(0,0,0))

    # Camera setting
    my_cam1 = Camera('cam1')
    self.my_camera1 = render.attachNewNode(my_cam1)
    self.my_camera1.setName('camera1')
    self.my_camera1.setPos(0, -0, 1)
    self.my_camera1.reparentTo(self.robot.chassisNP)

    my_cam2 = Camera('cam2')
    self.my_camera2 = render.attachNewNode(my_cam2)
    self.my_camera2.setName('camera2')
    self.my_camera2.setPos(0, 0, 70)
    self.my_camera2.lookAt(0,0,0)

    dr = base.camNode.getDisplayRegion(0)
    dr.setActive(0)
    window = dr.getWindow()

    w, h = self.robot.CAMX*2, self.robot.CAMY
    props = WindowProperties()
    props.setSize(w, h)
    window.requestProperties(props)

    self.dr1 = window.makeDisplayRegion(0, 0.5, 0, 1)
    self.dr1.setSort(dr.getSort())
    self.dr2 = window.makeDisplayRegion(0.5, 1, 0, 1)
    self.dr2.setSort(dr.getSort())

    self.dr1.setCamera(self.my_camera1)
    self.dr2.setCamera(self.my_camera2)

    # Light
    alight = AmbientLight('ambientLight')
    alight.setColor(Vec4(0.9, 0.9, 0.9, 1))
    alightNP = render.attachNewNode(alight)
    alightNP.setPos(0,0,5)

    dlight = DirectionalLight('directionalLight')
    dlight.setColor(Vec4(0.9, 0.9, 0.9, 1))
    dlightNP = render.attachNewNode(dlight)
    dlightNP.setPos(0,0, 5)

    render.clearLight()
    render.setLight(alightNP)
    render.setLight(dlightNP)

    self.setup()

    self.walls(Point3(0,18,0), Point3(20, 0.1, 5), Point3(1, 0.05, 2.5),
Point3(-1,-9,0), BulletBoxShape(Vec3(10, 0.01, 5)))
    self.walls(Point3(0,-18,0), Point3(20, 0.1, 5), Point3(1, 0.05, 2.5),
Point3(-1,-9,0), BulletBoxShape(Vec3(10, 0.01, 5)))
    self.walls(Point3(-18,0,0), Point3(0.1, 20, 5), Point3(0.05, 1, 2.5),
Point3(-10,-1,0), BulletBoxShape(Vec3(0.01, 10, 5)))
    self.walls(Point3(18,0,0), Point3(0.1, 20, 5), Point3(0.05, 1, 2.5),
Point3(-10,-1,0), BulletBoxShape(Vec3(0.01, 10, 5)))

    taskMgr.add(self.update, 'updateWorld')

    self.distance = 1000

    # Input
    '''inputState.watchWithModifiers('forward', 'w')
inputState.watchWithModifiers('left', 'a')
inputState.watchWithModifiers('reverse', 's')
inputState.watchWithModifiers('right', 'd')'''

def processInput(self, dt):
    force = Vec3(0, 0, 0)

```

```

torque = Vec3(0, 0, 0)

if inputState.isSet('forward'): force.setY( 0.5)
if inputState.isSet('reverse'): force.setY(-0.5)
if inputState.isSet('left'):   force.setX(-0.5)
if inputState.isSet('right'):  force.setX( 0.5)

force *= 30.0
torque *= 10.0

force = render.getRelativeVector(self.boxNP, force)
torque = render.getRelativeVector(self.boxNP, torque)

self.boxNP.node().setActive(True)
self.boxNP.node().applyCentralForce(force)
self.boxNP.node().applyTorque(torque)'''

def update(self, task):
    dt = globalClock.getDt()
    world.doPhysics(dt, 50, 0.008)

    '''self.processInput(dt)'''

    if not self.ctrl.stop():
        self.ctrl.update()
    else:
        return
    return task.cont

def setup(self):
    shape = BulletPlaneShape(Vec3(0, 0, 1), 1)
    node = BulletRigidBodyNode('Ground')
    node.addShape(shape)
    np = render.attachNewNode(node)
    np.setPos(0, 0, -1.5)
    world.attachRigidBody(node)

    # Box (dynamic)
    '''shape = BulletBoxShape(Vec3(0.5, 0.5, 0.5))
    self.boxNP = render.attachNewNode(BulletRigidBodyNode('Box'))
    self.boxNP.node().setMass(0.7)
    self.boxNP.node().addShape(shape)
    self.boxNP.setPos(0, 7, 1)
    tex = loader.loadTexture("textures/red.jpg")
    self.boxNP.setTexture(tex, 1)
    self.boxNP.setCollideMask(BitMask32.allOn())
    world.attachRigidBody(self.boxNP.node())'''

    visualNP = loader.loadModel('models/box.egg')
    visualNP.clearModelNodes()
    visualNP.reparentTo(self.boxNP)'''

def collide(self, collEntry):
    #print('Collision distance', collEntry.getIntoNode().getName())
    self.distance = float(collEntry.getIntoNode().getName())
    self.ctrl.commands[self.ctrl.count].flag = True

def initCollisionWall(self, obj, show, dist, center):
    collWallStr = 'CollisionWall' + " " + obj.getName()
    cNode = CollisionNode(collWallStr)
    cNode.addSolid(CollisionBox(center, dist))
    cNodepath = obj.attachNewNode(cNode)
    if show:
        cNodepath.show()
    return (cNodepath, collWallStr)

def initCollisionSphere(self, obj, show, center, num=0):
    collSphereStr = str(center[1]*30)
    #print (collSphereStr)
    cNode = CollisionNode(collSphereStr)
    cNode.addSolid(CollisionSphere (center, 0.2))
    cNodepath = obj.attachNewNode(cNode)
    if show:
        cNodepath.show()
    return (cNodepath, collSphereStr)

```

```

def walls (self, pos, scale, wall1, wall2, shape):
    tex = loader.loadTexture("textures/wall.jpg")
    node = BulletRigidTreeNode('Box')
    node.setMass(0)
    node.addShape(shape)
    np = render.attachNewNode(node)
    np.setPos(pos)
    world.attachRigidBody(node)
    self.box1 = loader.loadModel("cube")
    self.box1.setScale(scale)
    self.box1.reparentTo(np)
    self.box1.setTexture(tex, 1)

    tColl = self.initCollisionWall( self.box1, False, wall1, wall2)
    base.cTrav.addCollider(tColl[0], self.colHandEvent)

def take_screenshot(self, CAMX, CAMY):
    screen = self.drl.getScreenshot()
    file_name = 'results/res.jpg'
    screen.write(file_name)
    img = Image_Processing(file_name, CAMX, CAMY)
    return img.coord()

worldNP = render.attachNewNode('World')
world = BulletWorld()
world.setGravity(Vec3(0, 0, -9.81))

```

main.py

```

# --- Global libraries ---
import time
from Controller import ControllerInit
from Controller import ControllerForward
from Controller import ControllerTurn
from Controller import ControllerSequence
from Controller import ControllerFollow
from Controller import ControllerLearn
from Controller import ControllerForwardSmart

from panda3d.core import *
from panda3d.bullet import *

import numpy
import threading

class Option:
    def __init__(self, option):
        self.option = option

    def setup(self):
        if self.option == "a":
            # --- Import local libraries ---
            import direct.directbase.DirectStart
            from RobotModel3D import Robot
            from main_simulation3d import Simulation

            self.sim = Simulation()
            j=1 # For the number of a collision sphere
            for i in numpy.arange(0, COLLISION_DIST/30, 0.5):
                self.sim.sColl = self.sim.initCollisionSphere(self.
.sim.robot.robotModel, False, Point3(0,(COLLISION_DIST/30)-i+1,1),j)
                base.cTrav.addCollider(self.sim.sColl[0], self.sim.colHandE-
vent)
                self.sim.accept('into-' + self.sim.sColl[1], self.sim.collide)

```

```

        j+=1
        self.robot = self.sim.robot

        return self.robot

    else:
        # --- Import local libraries ---
        from easygopigo3 import EasyGoPiGo3
        from RobotDexter import Dexter
        gpg = EasyGoPiGo3()
        self.robot = Dexter(gpg)
        return self.robot

def run(self, sequence):
    self.sequence = sequence
    if self.option == "a":
        self.sim.ctrl = ControllerSequence(self.sequence)
        self.sim.ctrl.start()
        base.run()
    else:
        ctrl = ControllerSequence(self.sequence)
        ctrl.start()
        while not ctrl.stop():
            ctrl.update()
            time.sleep(0.01)

# --- Global variables ---

COLLISION_DIST = 150
SPEED = 295

# --- Choose an option between 3D Simulation and Real World Action
while True:
    option = input ("Do you want to: A) Play simulation B) Control the robot.\n[a/b]? : ")
    if option in ['a', 'b']:
        break

opt_robot = Option(option)
robot = opt_robot.setup()

forward = ControllerForward(robot, SPEED, COLLISION_DIST)
turn = ControllerTurn(robot, SPEED, 25)
turn_ = ControllerTurn(robot, SPEED, -90)
follow = ControllerFollow(robot, SPEED, COLLISION_DIST)

learn = ControllerLearn(robot, "Q", SPEED)
forward_smart = ControllerForwardSmart (robot, learn, SPEED)

#sequence = [forward, turn_, forward, turn, forward, turn]
#sequence = [learn]
sequence = [forward_smart]

opt_robot.run(sequence)

robot.shutdown()

```

images.py

```

import numpy as np
import argparse
import cv2
import datetime
import sys

class Image_Processing:

    def __init__(self, image_name, CAMX, CAMY):
        self.cX = CAMX/2
        self.cY = CAMY/2

        self.image_name = image_name

```

```

image = cv2.imread(image_name)
image = self.equalize_hist(image)

# Finding binary regions of red
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

mask1 = cv2.inRange(hsv, (0,50,20), (5,255,255))
mask2 = cv2.inRange(hsv, (175,50,20), (180,255,255))
mask = cv2.bitwise_or(mask1, mask2) # Important

kernel = np.ones((5,5),np.uint8)
mask = cv2.erode(mask,kernel,iterations = 1)

# Find the biggest red region
(cnts, _) = cv2.findContours(mask.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)

try:
    for cnt in cnts:
        if cv2.contourArea(cnt)<CAMX/5: # Don't count the noise
            pass
        else:
            c = max(cnts, key = cv2.contourArea)
            res = np.mean(c, axis=0)
            res = res[0]
            self.cX = int(round(res[0]))
            self.cY = int(round(res[1]))
except : # Empty one
    self.cX = CAMX/2
    self.cY = CAMY/2

# Equalizing the histogramm
def equalize_hist(self, img):
    img_yuv = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)
    img_yuv[:, :, 0] = cv2.equalizeHist(img_yuv[:, :, 0])
    img_output = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)
    return img

def coord(self):
    return self.cX, self.cY

```

Controller.py

```

import time
import math
import threading
import numpy as np
from numpy import random
import os
import matplotlib.pyplot as plt
from TrainingModel import EnvQLearning
from TrainingModel import EnvNN
from TrainingModel import EnvNNFollowColor
from TrainingModel import NeuralNetwork
from TrainingModel import NeuronLayer

class ControllerInit:
    'Initial state'
    def __init__(self, robot):
        self.robot = robot

    def start(self):
        pass

    def stop(self):
        pass

    def update(self):
        pass

class ControllerForward:

```

```

'Politics to move forward'
def __init__(self, robot, speed = 300, dist = 150):
    self.speed = speed
    self.dist = dist
    self.robot = robot
    self.flag = False

def start(self):
    self.robot.reset()
    self.flag = False
    self.robot.count = 1
    t = threading.Timer(0.5, self.robot.get_image)
    t.start()
    t.join()

def stop(self):
    return self.robot.condition(self)

def update(self):
    #print (self.robot.get_dist())
    cl, cr = self.robot.odometry()

    if self.stop():
        self.robot.shutdown()
        return
    self.robot.set_speed(self.speed*cl, self.speed*cr)

class ControllerTurn:
    'Politics to turn'
    def __init__(self, robot, speed = 300, angle = 90):
        self.speed = speed
        self.angle = angle
        self.robot = robot

    def start(self):
        self.robot.reset()
        #t = threading.Timer(0.5, self.robot.get_image)
        #t.start()
        #t.join()

    def angle_reached(self):
        res = self.robot.get_offset()
        offset = max(abs(res[1]), abs(res[0]))
        turn = ((self.robot.WHEEL_CIRCUMFERENCE*offset)/(self.robot.WHEEL_BASE_-CIRCUMFERENCE))/2

        return abs(turn)>=abs(self.angle)

    def stop(self):
        return self.angle_reached()

    def update(self):
        if self.stop():
            return

        if self.angle>0:                      # Turn right
            self.robot.set_speed(0, self.speed)
        else:                                # Turn left
            self.robot.set_speed(self.speed, 0)

class ControllerSequence:
    'Sequence of commands'
    def __init__(self, commands = []):
        self.commands = []
        self.commands = [x for x in commands]
        self.count = 0

    def start(self):
        self.count = -1

    def stop(self):
        return self.count >= len(self.commands)

    def update(self):
        if self.stop():

```

```

        return
    if self.count < 0 or self.commands[self.count].stop():
        self.count+=1
        if self.stop():
            return
        self.commands[self.count].start()
    self.commands[self.count].update()

class ControllerFollow:
    'Politics to follow an object'
    def __init__(self, robot, speed = 300, dist = 150):
        self.speed = speed
        self.dist = dist
        self.robot = robot
        self.cX = self.robot.CAMX/2
        self.cY = self.robot.CAMY/2
        self.flag = False
        self.taking_photo = True

    def start(self):
        t = threading.Thread(target=self.image, daemon = True)
        t.start()

        self.robot.reset()
        self.flag = False
        self.robot.count = 1

    def image(self):
        while self.taking_photo:
            self.cX, self.cY = self.robot.get_image()

    def stop(self):
        return self.robot.condition(self)

    def update(self):
        cl = 1
        cr = 1
        diff = int(self.robot.CAMX/10)
        if self.cX < (self.robot.CAMX/2 - diff):
            cl = 0.5
            cr = 1.5
        elif self.cX > (self.robot.CAMX/2 + diff):
            cl = 1.5
            cr = 0.5
        else:
            cr = 1
            cl = 1

        if self.stop():
            self.taking_photo = False
            self.robot.shutdown()
            return
        self.robot.set_speed(self.speed*cl, self.speed*cr)

class ControllerLearn:
    'Training'
    def __init__(self, robot, option, speed = 300):
        self.speed = speed
        self.robot = robot
        self.option = option
        self.taking_photo = True
        self.cX, self.cY = self.robot.CAMX/2, self.robot.CAMY/2

    def image(self):
        while self.taking_photo:
            self.cX, self.cY = self.robot.get_image()

    def start(self):
        if self.option == "Q":
            self.env = EnvQLearning(self)
        if self.option == "NN":
            #self.env = EnvNN(self)
            t = threading.Thread(target=self.image, daemon = True) #or put it
outside of controllers
            t.start()

```

```

        self.env = EnvNNFollowColor(self)
        self.env.train()
        f = open("weights.txt", "w+")
        f.write(str(self.env.neural_network.layer1.weights))
        f.write(str(self.env.neural_network.layer2.weights))
        f.close()

        self.robot.reset()
        self.k = 0
        self.end_episode = False
        self.stop_simulation = False

    def stop(self):
        return self.stop_simulation

    def update(self):
        if self.end_episode:
            self.env.reward_list.append(self.env.cumulated_reward)
            self.env.episode_list.append(int(self.k+1))
            self.env._reset()

        if self.k >= self.env.epochs:
            print ('GAME OVER')
            self.stop_simulation = True
            print (self.env.reward_list)
            print (self.env.episode_list)
            plt.plot(self.env.episode_list, self.env.reward_list)
            plt.ylabel('reward')
            plt.xlabel('episode')
            plt.show()

        return
        self.env._update()

class ControllerForwardSmart:
    'Testing the ANN'
    def __init__(self, robot, ctrl, speed = 300):
        self.robot = robot
        self.speed = speed
        self.ctrl = ctrl
        self.k = 0
        self.taking_photo = True
        self.cX, self.cY = self.robot.CAMX/2, self.robot.CAMY/2

    def image(self):
        while self.taking_photo:
            self.cX, self.cY = self.robot.get_image()

    def start(self):
        # --- Approach the wall ---
        #self.env = EnvNN(self)
        #layer1 = NeuronLayer(4, 1) # 4 neurons, 1 input
        #layer2 = NeuronLayer(1, 4) # output
        #layer1.weights = [[9.33508627, -0.14762897, -13.5186365, 0.97483954]]
        #layer2.weights = [[ 8.49960835], [ -1.6299063], [-17.82664448],
        [-0.43608188]]
        #self.env.neural_network = NeuralNetwork(layer1, layer2)

        # --- Follow Color ---
        t = threading.Thread(target=self.image, daemon = True)
        t.start()

        self.env = EnvNNFollowColor(self)

        layer1 = NeuronLayer(4, 1) # 4 neurons, 1 input
        layer2 = NeuronLayer(2, 4) # output
        layer1.weights = [[1.42118996, -1.92753475, -5.08044704, 1.13369032]]
        layer2.weights = [[ -3.43629242, -4.35311457],
                          [ 24.65651608, 36.16606826],
                          [-31.92160833, -25.99358137],
                          [ -0.50843814, -6.09702519]]
        self.env.neural_network = NeuralNetwork(layer1, layer2)

```

```

def stop(self):
    dist = self.robot.get_dist()
    if self.k >= self.env.epochs or dist < 100:
        self.taking_photo = False
    return True

def update(self):
    self.env._update()

```

TrainingModel.py

```

import numpy as np
import time
from numpy import exp, array, random, dot
import threading

class EnvQLearning:
    def __init__(self, ctrl):
        self.ctrl = ctrl
        self.actions = [0, 1, 2, 3] #speed values
        self.states = [0, 1, 2, 3, 4, 5] #distance from the wall, 0 = far, 5 =
close
        self.stateCount = len(self.states)
        self.actionCount = len(self.actions)
        self.stop_count = 0

        # hyperparameters
        self.epochs = 15
        self.gamma = 0.1
        self.epsilon = 0.01
        self.decay = 0.1

        self.state = 0
        self.reward = 0

        self.qtable = np.random.rand(self.stateCount, self.actionCount).tolist()

    def _reset(self):
        time.sleep(0.5)
        print ('-----RESET-----')
        self.ctrl.robot.chassisNP.setPos(0, 0, 0)
        self.ctrl.robot.set_speed(0,0)
        self.ctrl.k+=1
        self.ctrl.robot.sim.distance = 1000
        self.ctrl.robot.count = 1 # Speed factor
        self.stop_count = 0
        self.epsilon -= self.decay*self.epsilon

        return 0, 0, False

    def step(self, action):
        done = False
        self.dist_value = self.ctrl.robot.get_dist()

        print ('Distance is '+str(self.dist_value))
        if action==0: # Full speed
            print ('Full speed')
            self.ctrl.robot.set_speed(self.ctrl.speed, self.ctrl.speed)
        if action==1: # Half-speed
            print ('speed 0.8')
            self.ctrl.robot.set_speed(self.ctrl.speed*0.8, self.ctrl.speed*0.8)
        if action==2: # Quarter-speed
            print ('speed 0.5')
            self.ctrl.robot.set_speed(self.ctrl.speed*0.5, self.ctrl.speed*0.5)
        if action==3: # Stop
            print ('Full stop')
            self.ctrl.robot.set_speed(0, 0)

        # Reward table
        if self.dist_value == 60:
            reward = 5
        elif self.dist_value == 75:

```

```

        reward = 1
    elif self.dist_value == 90:
        reward = 0
    elif 120 >= self.dist_value >= 105:
        reward = 0
    elif self.dist_value == 45:
        reward = -100
    else:
        reward = -1

    # Choosing next state?..
    if self.dist_value == 60:
        nextState = 4
        self.stop_count +=1
        if self.stop_count>100:
            done = True
    elif self.dist_value == 75:
        nextState = 3
    elif self.dist_value == 90:
        nextState = 2
    elif self.dist_value == 105:
        nextState = 1
    elif self.dist_value == 45:
        nextState = 5
        done = True
    else:
        nextState = 0

    return nextState, reward, done

def randomAction(self):
    return np.random.choice(self.actions)

def _update(self):
    if np.random.uniform() < self.epsilon:
        action = self.randomAction()
    else:
        action = self.qtable[self.state].index(max(self.qtable[self.state]))

    next_state, self.reward, self.ctrl.end_episode = self.step(action) # take action
    self.qtable[self.state][action] = self.reward + self.gamma * max(self.qtable[next_state]) # update qtable
    self.state = next_state # update state

class NeuronLayer():
    def __init__(self, number_of_neurons, number_of_inputs_per_neuron):
        self.weights = 2 * random.random((number_of_inputs_per_neuron, number_of_neurons)) - 1

class NeuralNetwork():
    def __init__(self, layer1, layer2):
        self.layer1 = layer1
        self.layer2 = layer2

    def __sigmoid(self, x): #normalize 1-0
        return 1 / (1 + exp(-x))

    def __sigmoid_derivative(self, x): #gradient
        return x * (1 - x)

    def train(self, training_set_inputs, training_set_outputs, number_of_training_iterations):
        for iteration in range(number_of_training_iterations):

            output_layer_1, output_layer_2 = self.forward(training_set_inputs)

            layer2_error = training_set_outputs - output_layer_2
            layer2_delta = layer2_error * self.__sigmoid_derivative(output_layer_2)

            layer1_error = layer2_delta.dot(self.layer2.weights.T)
            layer1_delta = layer1_error * self.__sigmoid_derivative(output_layer_1)

```

```

        layer1_adjustment = training_set_inputs.T.dot(layer1_delta)
        layer2_adjustment = output_layer1.T.dot(layer2_delta)

        self.layer1.weights += layer1_adjustment
        self.layer2.weights += layer2_adjustment

    def forward(self, inputs):
        output_layer1 = self.__sigmoid(dot(inputs, self.layer1.weights))
        output_layer2 = self.__sigmoid(dot(output_layer1, self.layer2.weights))
        return output_layer1, output_layer2

class EnvNN:
    def __init__(self, ctrl):
        self.ctrl = ctrl
        self.epochs = 100
        self.ctrl.robot.reset()

        t = threading.Thread(target=self.ctrl.image, daemon = True) #or put it
outside of controllers
        t.start()

    def train(self):
        layer1 = NeuronLayer(4, 1) # 4 neurons, 1 input
        layer2 = NeuronLayer(1, 4) # output

        self.neural_network = NeuralNetwork(layer1, layer2)

        training_set_inputs = array([[15.0],[30.0], [60.0], [75.0], [90.0],
[150.0], [300], [1000.0]])
        training_set_inputs = self.normalize(training_set_inputs)

        training_set_outputs = array([[ 0, 0.1, 0.2, 0.5, 0.7, 0.95, 0.99,
1]]).T

        self.neural_network.train(training_set_inputs, training_set_outputs,
50000)

    def normalize(self, arr):
        res = arr
        for i in range(len(arr)):
            _res[i][0] = (arr[i][0] - min(arr))/(max (arr) - min(arr))
        return _res

    def calculate_speed(self, value):
        new_input = (value - 15.0)/(1000.0 - 15.0)
        hidden_state, output = self.neural_network.forward(array([new_input]))
        return (output[0])

    def update(self):
        dist_value = self.ctrl.robot.get_dist()
        self.calculate_speed(self.ctrl.cX)
        new_speed = self.calculate_speed(self.ctrl.cX)
        if new_speed < 0.15:
            self.ctrl.k+=1
        self.ctrl.robot.set_speed(self.ctrl.speed*new_speed,
self.ctrl.speed*new_speed)

class EnvNNFollowColor:
    def __init__(self, ctrl):
        self.ctrl = ctrl
        self.epochs = 100
        self.ctrl.robot.reset()

    def train(self):
        layer1 = NeuronLayer(4, 1) # 4 neurons, 1 input
        layer2 = NeuronLayer(2, 4) # output

        self.neural_network = NeuralNetwork(layer1, layer2)

        training_set_inputs = array ([[320.0/2], [1.0], [60.0/2],[580.0/2],
[640.0/2]])
        training_set_inputs = self.normalize(training_set_inputs)

```

```

    training_set_outputs = array([[1.0, 1.0], [0.0, 0.6], [0.0, 0.5], [0.5,
0.0], [0.6, 0.0]])
    self.neural_network.train(training_set_inputs, training_set_outputs,
50000)

    def normalize(self, arr):
        res = arr
        for i in range(len(arr)):
            res[i][0] = (arr[i][0] - min(arr))/(max (arr) - min(arr))
        return res

    def calculate_speed(self, value):
        new_input = (value - 1.0)/(640.0/2 - 1.0)
        hidden_state, output = self.neural_network.forward(array([new_input]))
        return output

    def _update(self):
        self.calculate_speed(self.ctrl.cX)
        new_speed = self.calculate_speed(self.ctrl.cX)
        self.ctrl.robot.set_speed(self.ctrl.speed*new_speed[0],
self.ctrl.speed*new_speed[1])

```