Sistema de navegación para robot móvil basado en aprendizaje por refuerzo

César Daniel Garrido Urbano, Fernando Lozano Martínez, Carolina Higuera Arias

***Resumen*—En este proyecto de grado se aborda el problema de navegación para robots móviles utilizando aprendizaje por refuerzo profundo (*Deep Reinforcement Learning* o DRL). De forma especifica, se implementan algunas variantes de *Deep Q-Learning* para entrenar a un robot móvil en la tarea de navegar hacia las persona que detecta (a través de una cámara RGB) en simulación. Para esto, se plantea el problema de navegación dentro del contexto de aprendizaje por refuerzo y se comparan los algoritmos de *Double DQN* [1] y *Clipped Double Q Learning* [2] a través de una implementación propia. Como caso de estudio se utiliza la plataforma *Pepper*, un robot semi-humanoide con base omnidireccional diseñado por *Softbank Robotics* para la interacción con humanos. Actualmente, el robot es capaz de realizar detección de rostros e incorpora sensores como cámaras RGB, cámara de profundidad y láseres, los cuales se utilizan como entrada al sistema de navegación propuesto. Adicionalmente, se utiliza la herramienta de simulación *qiBullet* [3] para realizar el entrenamiento y validar los resultados de las mejores políticas. El comportamiento del robot con el sistema de navegación desarrollado se puede observar en el siguiente enlace: https://youtu.be/ncqNx_Q3TUg**

***Index Terms*—Aprendizaje por refuerzo, Deep Q-Learning, *qiBullet*, Navegación, Robot Pepper.**

I. Introducción

La navegación es uno de los problemas fundamentales y más estudiados en el área de la robótica móvil. Las soluciones tradicionales a este problema consisten en varios sub-sistemas que realizan distintas tareas, como las de mapeo, localización y planeación de ruta, entre otras [4]. El trabajo conjunto de estas tareas le permiten a los robots localizarse dentro de un mapa (la mayoría de veces predefinido o construido con anterioridad) y navegar hasta su destino evitando obstáculos. No obstante, estas soluciones tienden a depender mucho del entorno y suelen necesitar el ajuste cuidadoso de una gran número de parámetros para tener el desempeño deseado [5] [6]. Esto se convierte en un problema aun mayor cuando el robot debe navegar en entornos cambiantes (como ambientes sociales) o el hardware no permite una correcta o completa observación del mismo [7] [8]. Razón por la cual se ha empezado a implementar técnicas de *Machine Learning* en el área de navegación, las cuales han demostrado resultados favorables, especialmente con el uso de técnicas de aprendizaje por refuerzo [6] [9] [10] [11].

La universidad de los Andes, a través de la Alianza SinfonIA, cuenta con el robot social tipo *Pepper* de *Softbank Robotics*. Este robot semi-humanoide y omnidireccional, cuenta con distintos tipos de sensores como cámaras RGB, de profundidad, sensores infrarrojos, sonares, etc. y está orientado a la interacción con humanos. Actualmente, el robot es capaz de reconocer rostros y estimar su distancia con este a través del procesamiento de las imágenes adquiridas con sus cámaras RGB [12] . No obstante, su capacidad de procesamiento y la calidad de los sensores que posee han hecho que la tarea de navegación autónoma sea un reto importante [5]. A pesar de que el robot cuenta con un sistema de navegación tradicional (basado en SLAM), se desearía tener un sistema robusto que le permita al robot navegar hasta las personas que logre detectar, ya sea en conjunto o en reemplazo del sistema que actualmente utiliza.

De esta manera, en este proyecto de grado se exploraron e implementaron algunas de las técnicas de aprendizaje por refuerzo para resolver el problema propuesto. Para esto se definió el problema de navegación como un proceso de desición de Markov (MDP), es decir se definieron elementos como los estados, el espacio de acciones, las recompensas, etc. Posteriormente, se utilizó la herramienta de *qiBullet* [3] para simular el entorno deseado y sobre este implementar algoritmos de aprendizaje por refuerzo profundo (o DRL por sus siglas en ingles) como *DQN* [13], *Double DQN* [1] y *Clipped Double Q-Learning* [2]. Para esto se hizo tanto una implementación propia como una implementación con la librería de aprendizaje por refuerzo *Stable Baselines* [14]. Se realizaron los entrenamientos respectivos, ajustando los distintos parámetros del aprendizaje como: Arquitectura del modelo, entorno, recompensa, tasa de aprendizaje, tasa de descuento, tasa de exploración, entre otros. Finalmente, se seleccionaron los mejores modelos para los distintos algoritmos implementados y se validaron en simulación.

II. Estado del Arte

Las primeras soluciones al problema de navegación autónoma se remontan a las propuestas de mapeo y localización simultanea (SLAM) al inicio de la década de los noventas [15] [16]. Hoy en día estas técnicas (o variaciones de las mismas) se han convertido en el estándar de la navegación autónoma y la robótica móvil [8]. No obstante, en años recientes, las técnicas de *Machine Learning* han ganado popularidad, dado su gran desempeño y facilidades en implementación con el incremento de la capacidad computacional. Actualmente, existen varios trabajos que incorporan dichas técnicas como solución a uno o varios de los problemas comunes en el área de la navegación.

Aunque existen precedentes de soluciones con aprendizaje supervisado como en [17] y en [18] (las cuales se enfocan principalmente en el problema de planeación de ruta), el aprendizaje por refuerzo ha sido una de las técnicas de *Machine Learning* que más resultados ha brindado en esta área. Con esta aproximación no se requiere de datos etiquetados los cuales son especialmente difíciles de adquirir en este tipo de problemas. Especialmente, si se desea aprovechar las capacidades de modelos profundos los cuales requieren todavía de una mayor cantidad de datos para su entrenamiento. A continuación se presentan algunos de los trabajos más recientes, relacionados con esta área, en donde el aprendizaje por refuerzo profundo ha brindado resultados prometedores.

En [9] se propone y se implementa el algoritmo D3QN (el cual incorpora *Dueling DQN* y *Double Q-Learning*) para desarrollar un sistema de navegación sobre un robot móvil en el simulador de Gazeboo. Este le permite al agente navegar hasta un objetivo fijo a través de únicamente sus cámaras RGB. En [10] se presenta un sistema de evasión de obstáculos, basado en *Double DQN*, en donde el agente navega hasta los sub-objetivos de la planeación de ruta global. Para esto el agente toma como entrada su posición, la del objetivo y los obstáculos detectados. Este sistema, que funciona en conjunto con uno tradicional, fue entrenado en simulación y validado en físico. Por su parte, en [11], se presenta un sistema de navegación basado en visión, entrenado con el algoritmo de A2C (con algunas mejoras realizadas para las tarea de navegación). En este el agente recibe como entrada únicamente la imagen RGB que observa y la de su objetivo, desarrollo realizado en la herramienta de simulación AI2THOR. Finalmente, en [19] se presenta un algoritmo auto-supervisado con grafos de computación generalizada que considera métodos basados en valor (*value-based model-free*) y métodos basados en modelos (*model-based*). Este modelo aprende directamente de imágenes y fue validado en físico, donde al agente le tomó tan solo algunas horas aprender a navegar de forma adecuada.

Ahora bien, aunque existe un gran número de desarrollos en esta área sobre robots móviles, en lo que respecta al robot del caso de estudio (*Pepper*), los desarrollos en navegación se han hecho principalmente sobre sistemas tradicionales. Ya sea en busca de validar su desempeño con el ajuste de parámetros como en [5] o en la utilización de sus cámaras para realizar un Visual-SLAM como en [7]. Y, aunque han habido varios desarrollos en la incorporación de DRL sobre esta plataforma, estos han sido enfocados directamente a la interacción social [20] o a la manipulación de objetos [21]. Hasta ahora solo en [6] se ha realizado una implementación de DRL para la navegación sobre el robot *Pepper*. En este trabajo Leiva et. al proponen un sistema de evasión de obstáculos, parametrizado por una red neuronal profunda con entrada multimodal (que incorpora información de lásers, cámara de profundiad y odometria), la cual se entrena con el algoritmo de *Deep Deterministic Policy Gradient* o DDPG con el simulador *Gazeboo* y se prueba en físico con resultados bastante sobresalientes.

III. Marco teórico y conceptual

III-A. Generalidades de los sistemas de navegación

En la actualidad los sistemas de navegación se pueden clasificar en las siguientes 4 categorías [4]:

- **Map based:** Enfoque tradicional en donde se utiliza un mapa como referencia para la navegación. Este requiere típicamente de tres tareas muy especificas: Localización, mapeo y planeación de ruta.
- **Behaviour based:** Enfoque basado en tareas especificas o predeterminadas como: Evasión de obstáculos, seguir una pared o buscar un objetivo, entre otros. Estos problemas se suelen resolver con una acción dinámica de control, pero suele tener problemas cuando la complejidad de estas tareas es alta.
- **Learning based:** Enfoque que utiliza técnicas de *Machine Learning* para que un agente pueda aprender a resolver un problema de navegación especifico.
- **Communication based:** Enfoque basado en la comunicación de información sensada por varios agentes, control descentralizado y coordinación.

En este proyecto de grado se desarrolla un sistema de navegación *learning based*, con el objetivo de cumplir una tarea especifica que bien podría caer dentro de la categoría de *behaviour based*. Pues se desea que el robot encuentre y navegue hacia un objetivo especifico, el cual para este caso son las personas que detecte con su cámara RGB.

III-B. Aprendizaje por refuerzo

El aprendizaje por refuerzo (o RL por sus siglas en ingles) es un área del *Machine Learning* con la que se busca enseñarle a un agente (máquina) una tarea especifica a través de su interacción con el entorno. Esto difiere del enfoque más común de aprendizaje supervisado donde la máquina aprende a partir de datos de entrenamiento etiquetados (supervisados). El esquema general del aprendizaje por refuerzo se presenta en la figura 1. En este se tiene un agente que es capaz de observar el entorno a través de lo que se denominan estados (S), interactúa con este a través de acciones (A) y sabe si su interacción es correcta o no a través de recompensas (R), las cuales pueden ser positivas o negativas.

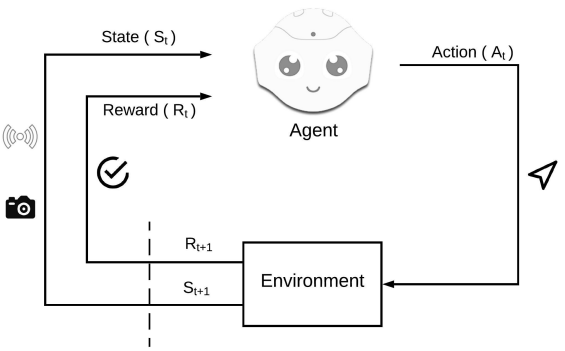

Figura 1: Funcionamiento general del aprendizaje por refuerzo

De esta manera a lo largo del entrenamiento el agente toma acciones (a_t) con base en el estado que esta observando (s_t), lo que posteriormente lo lleva a un nuevo estado (s_{t+1}) y le permite percibir una recompensa de esta interacción (r_t), proceso que se repite de forma iterativa y que en últimas busca que el agente aprenda comportamientos que maximicen la recompensa acumulada a través del tiempo. A estos comportamientos se les denomina política (π) y para encontrar o estimar la política óptima se han desarrollado distintos algoritmos. De forma especifica en este proyecto de grado el trabajo se enfocó en los algoritmos de *Q-Learing*, incluyendo algunas de sus variantes, las cuales se detallan brevemente a continuación:

III-B1. Q-Learning: Esta es una técnica de RL, basada en diferencias temporales (TD), introducida a finales de la década de los ochenta por Watkins [22]. Las técnicas de TD pueden aprender directamente de experiencias con el entorno sin un modelo de este, como las técnicas de Monte Carlo y además actualizan sus estimativos a partir de otros estimativos (con bootstraping) sin esperar al resultado final, como en programación dinámica [23, chapter 6]. En términos generales, *Q-Learning* busca estimar una función de valor-acción conocida como Q(s,a), la cual representa el valor esperado de la recompensa acumulada para ese par estado-acción (s,a). De esta manera, si se conoce el valor de la recompensa para cada par estado-acción, es posible maximizar la recompensa del agente siguiendo la serie de acciones que maximizan la función Q. Para esto, *Q-Learning* actualiza los valores Q de la siguiente manera, actualización que se deriva de la ecuación de Bellman:

$$\begin{aligned} Q(S_t, A_t) \leftarrow & Q(S_t, A_t) + \\ & \alpha[r_t + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \end{aligned} \tag{1}$$

Donde:

- r es la recompensa percibida por tomar la acción A_t en el estado S_t.
- α es la tasa de aprendizaje (importancia que se le da a cada actualización).
- γ es la tasa de descuento (*tradeoff* entre recibir recompensas inmediatas o a largo plazo).

Adicionalmente, dado que *Q-Learning* aproxima directamente la función de valor, independiente de la política (π) que se este usando, esta se conoce como una técnica de control *off-policy*. Esto permite utilizar acciones exploratorias a través de políticas $\epsilon - greedy$. De esta manera, se puede ajustar el parámetro ϵ para determinar la probabilidad de que el agente explore el entorno (tomando una acción aleatoria) o explote (sea *greedy* con respecto a) la función Q estimada.

III-B2. Deep Q-Learning & DQN: Ahora bien, la propuesta original de *Q-Learning* se implementa sobre su versión tabular, para un espacio de estados y acciones discreto. Es decir la función Q se aproxima a través de una matriz con N filas y M columnas, donde N es el número de estados y M es el número de acciones. El problema con esta aproximación es que para poder asegurar convergencia se deben visitar y actualizar todos los pares estado-acción [23, chapter 6], algo que incrementa en dificultad con el aumento de estados y acciones. Por esta razón, para entradas complejas (como imágenes) era necesario realizar un preprocesamiento que permitiera extraer características y reducir el número de estados. Para atacar este problema, en 2013 el equipo de DeepMind propone el algoritmo de *Deep Q-Networks* (o DQN), una variante de *Q-Learning* que utiliza redes neuronales convolucionales para estimar la función Q a partir de entradas crudas de imágenes de juegos de Atari [13]. De esta forma, la actualización de la función Q no se hace a través de la ecuación de Bellman (1) sino con el error de Bellman al cuadrado (SBE):

$$SBE = \frac{1}{2}(r_t + \gamma max_a Q_\theta(S_{t+1}, a) - Q_\theta(S_t, A_t))^2 \tag{2}$$

En otras palabras, lo que se hace es calcular los valores Q objetivo Q_{target} y con estos entrenar una red neuronal con parámetros θ:

$$Q_{target} = r_t + \gamma max_a Q_\theta(S_{t+1}, a) \tag{3}$$

Vale la pena aclarar que para un estado terminal el Q_{target} corresponde solo a la recompensa percibida y no se considera el valor Q del estado siguiente.

Adicionalmente, la implementación de este algoritmo incluye algunas otras técnicas para hacer más eficiente y estable el proceso de entrenamiento, como lo son:

- *Experience Replay Buffer:* Se utiliza una memoria que guarda cada una de las transiciones (s_t, a_t, r_t, s_{t+1}) y se utiliza una muestra aleatoria (*mini-batch*) para entrenar la red neuronal en cada step. Adicionalmente, en [24] se introduce la idea de *Prioritize Experience Replay*, en donde en vez de seleccionar una muestra aleatoria se le agrega un peso a la experiencia (con base en el error de estimación) y se selecciona el *mini-batch* dando prioridad a aquellas transiciones más relevantes.
- *Target Network:* Se utiliza una copia de la red neuronal principal (más vieja) para computar los valores Q_{target}. Esto evita que la función Q cambie muy rápido y termine retroalimentandose al actualizar sus propios valores Q. De esta manera los valores Q objetivo se actualizan de la siguiente forma:

$$Q_{target} = r_t + \gamma max_{a'} Q_{\theta'}(S_{t+1}, a) \tag{4}$$

Donde $Q_{\theta'}(s, a)$ es la copia más vieja de la red neuronal principal (red objetivo o *target*) con parámetros θ'. Esta se actualiza cada *n* pasos con los pesos de la red neuronal principal (de parámetros θ), la cual se actualiza en cada paso del entrenamiento.

III-B3. Double DQN: Uno de los principales problemas con *Q-Learning* es la sobreestimación de valores Q, resultado de incorporar la función de maximización dentro de su actualización, a esto se le conoce como *maximization bias*

[23, chapter 6]. Como solución a esto, Van Hassel propone en 2010 el algoritmo de Double Q-Learning [25]. Este utiliza dos funciones Q distintas (Q_A y Q_B) y se actualizan (utilizando la ecuación 1) pero con los valores Q futuros de la otra función. Es decir, se utiliza la función Q_A para calcular la acción a^* que se debe tomar en el estado S_{t+1}, pero se evalúa sobre la función Q_B y viceversa. La clave de esta técnica radica en el desacople de la selección de la acción y la evaluación de la misma en dos distintos modelos. Por esto en 2015, Van Hassel et al. presentan una versión actualizada de Double Q-Learning para DQN (Double DQN) [1]. En esta versión utilizan la *target network* para hacer el desacople mencionado. De esta forma, la actualización de los valores Q se realiza se la siguiente forma:

$$Q_{target} = r_t + \gamma Q_\theta(S_{t+1}, argmax_{a'} Q_{\theta'}(S_{t+1}, a')) \quad (5)$$

En este caso se utiliza la *target network* para seleccionar la acción y la red neuronal principal para evaluar la acción. Adicionalmente, se propone una copia suave de parámetros entre la red neuronal principal y la objetivo a través del promedio de Polyak, donde se actualizan los pesos de la red objetivo en cada iteración de la siguiente manera:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \quad (6)$$

Donde τ es la tasa con la que se promedian los valores de los parámetros. Esto propone una mejora a la copia de parámetros cada n pasos, pues esta resulta en el mismo algoritmo de DQN al momento de igualar los parámetros de las redes neuronales.

III-B4. Clipped Double Q-Learning: Finalmente, una última propuesta como mejora al algoritmo de Double Q-Learning se hace en [2]. En esta Fujimoto et al. proponen utilizar la aproximación inicial de Van Hassel [25], en donde se utilizan dos modelos distintos de función Q, tomando como estimativo el menor valor Q del estado siguiente.

$$Q_{target} = r_t + \gamma min_{i=1,2}[Q_{\theta_i}(S_{t+1}, argmax_{a'} Q_{\theta'_i}(S_{t+1}, a'))] \quad (7)$$

Según los autores esto puede resultar en una subestimación de los valores Q, contrario a una sobreestimación que ocurre con el Q-learning tradicional e incluso con el Dobule DQN (cuando la red neuronal objetivo es muy parecida a la principal), lo que en últimas es una estimación más deseada.

IV. Caso de Estudio: Robot tipo *Pepper*

Pepper es un robot semi-humanoide fabricado por *SoftBanck Robotics* con un enfoque social, su diseño está orientado específicamente para la interacción con humanos. Para esto cuenta con distintos tipos de sensores como: Cámaras RGB, cámara de profundidad (3D), sensores infrarrojo, sonares, sensores de tacto, entre otros *(Véase figura* 2). Asimismo, cuenta con actuadores para el movimiento de varias de sus articulaciones como cuello, hombros, brazos, etc. y una base omnidireccional con tres ruedas, que le permite desplazarse en cualquier dirección, ya sea lineal (x, y) o rotacional (θ).

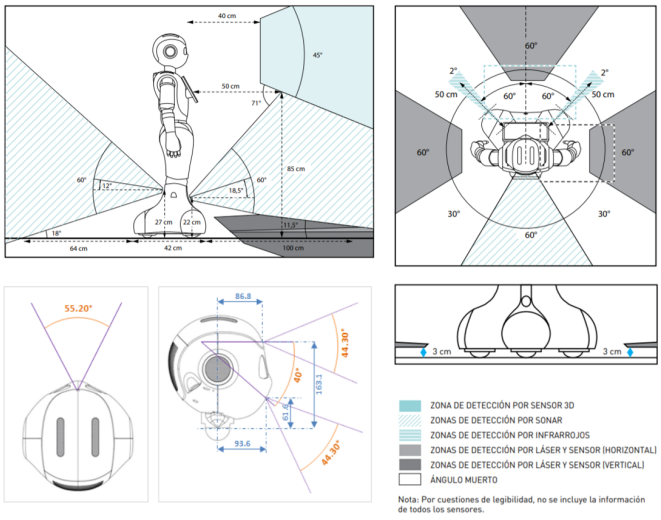

Figura 2: Rango de detección de algunos de los sensores del robot Pepper. Tomado de [26]

Adicionalmente, el robot incorpora la funcionalidad de detección de rostros con sus cámaras RGB. Esto lo puede hacer a través de su sistema operativo nativo (*NaoQi*) o a través de desarrollos realizados sobre su plataforma con el *framework* de ROS [12]. Estos últimos utilizan el algoritmo de *Viola Jones* para el reconocimiento de rostros a través de la librería de *OpenCV*. Con esto se puede saber el número de rostros en una imagen detectada con la cámara RGB del robot y los recuadros de los rostros dentro de esta (*Véase figura* 3). Esto brinda información de la posición relativa de la persona: Entre mayor sea el tamaño del recuadro, más cerca se encuentra de la persona y viceversa. De igual forma, si la persona se encuentra ligeramente a la derecha o a la izquierda esto se verá reflejado en la posición del recuadro dentro de la imagen.

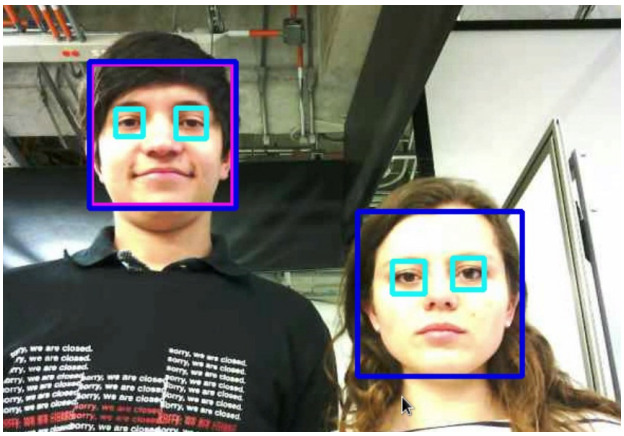

Figura 3: Recuadro de rostros que detecta el robot Pepper, con su cámara RGB, a partir de los desarrollos hechos en [12]

IV-A. Entorno de Simulación (qiBullet)

Para realizar el entrenamiento se utilizó la herramienta *qiBullet* [3], un simulador basado en *pybullet* que incorpora los modelos de los robots de *Softbank Robotics* como *Pepper*. En este se construyeron distintos entornos con los cuales interactúa el agente para entrenar y validar el sistema de navegación propuesto (véase figura 4). En este caso, para emular la detección de rostros, la cual no hace parte del alcance de este proyecto, se utilizó una máscara de color sobre las personas simuladas. Estas se modelaron con una esfera roja que hace las veces de cabeza, de forma que la máscara de color pueda detectarlas de forma análoga a como lo haría el sistema de detección de rostros en la práctica.

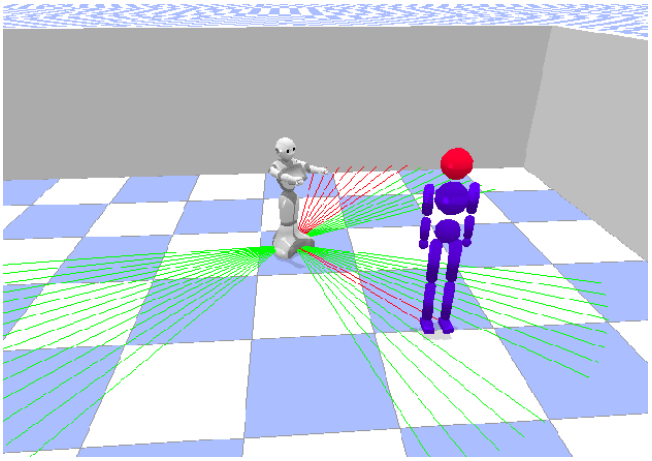

Figura 4: qiBullet, herramienta de simulación utilizada para modelar el entorno

Adicionalmente, con esta herramienta se construyeron dos tipos de escenarios: Un cuarto vacío, que le permitiera al agente a acercarse a las personas sin la necesidad de evadir obstáculos y otro con paredes intermedias, tratando de emular de mejor forma lo que sería un entorno de habitaciones. Sobre estos se delimitaron algunas áreas donde el robot podría iniciar cada uno de los episodios y áreas donde estarían ubicadas las personas de forma aleatoria (véase figura 5).

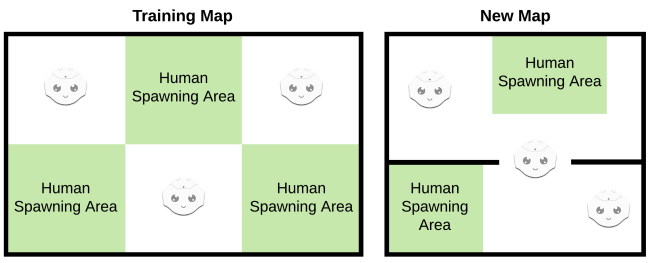

Figura 5: Mapas construidos para el entrenamiento

Todas las simulaciones se realizaron en un computador portátil con procesador Intel Core i7-8750H y una GPU Nvidia GeForce GTX 1050Ti. Los algoritmos de aprendizaje por refuerzo y los modelos de red neuronal se implementaron en Python con Tensorflow. Con esta capacidad computacional 1000 pasos de simulación tomaban aproximadamente una hora.

IV-B. Definición de los elementos de Aprendizaje por Refuerzo

Para poder plantear el problema de navegación dentro del contexto de aprendizaje por refuerzo es necesario definirlo como un ambiente estocástico descrito por un Proceso de Decisión de Markov (MDP). Esto es posible pues el entorno del caso de estudio satisface con la propiedad de Markov, la cual establece que la probabilidad de transición al estado S_{t+1}, depende únicamente de del estado actual S_t y la acción tomada a_t. En este caso especifico, se asume que los entornos son estáticos, es decir ni los obstáculos ni las personas están en movimiento. Con esto en mente, a continuación se describen los elementos del MDP y del aprendizaje por refuerzo definidos para el caso de estudio. Vale la pena aclarar que varios de estos elementos se fueron ajustando a lo largo del proyecto y se presentan únicamente con los que se realizaron las pruebas definitivas.

IV-B1. Estados: Con el objetivo de obtener observaciones más completas del entorno se utilizaron varios de los sensores que incorpora el robot Pepper. Para esto los estados se consideran como una entrada multimodal que incorpora información de la cámara de profundidad, las *features* o características del rostro detectado por la cámara RGB y los láseres. Así las cosas a continuación se presenta una descripción de cada una de las entradas que componen el estado (s_t):

- **Cámara de Profundidad (s_d^t):** Sensor 3D con resolución de 320x240, capaz de detectar distancia hasta un objeto en un rango de 40 grados a la altura del rostro del robot. Para esta implementación se redimensionó esta entrada a 80x60 como en [6].
- ***Features* del rostro de la cámara RGB (s_f^t):** Recuadro resultante de la detección de rostros con la cámara RGB. De este se utilizan únicamente dos variables: la posición horizontal del recuadro y el área del recuadro, ambos en términos de píxeles.
- **Lásers (s_l^t):** Tres sets de láseres, uno en dirección frontal y dos en dirección lateral (izquierda y derecha), cada uno con 15 puntos de detección de hasta 3 metros, para un vector final de 45 datos de entrada.

Vale la pena resaltar que las entradas descritas anteriormente fueron normalizadas y utilizadas en la implementación propia de los algoritmos de DQN, Double DQN y Clipped Double Q-Learning. Sin embargo, se realizaron algunas pruebas con la librería de aprendizaje por refuerzo *Stable Baselines* [14], en donde se utilizó el algoritmo de Double DQN. Para esto fue necesario hacer algunos ajustes a las entradas, dado que la librería las limita a un objeto de tipo *Box* con una dimensión predefinida. De esta manera, la única entrada que se mantuvo igual fue la de la imagen de profundidad. Para la entrada de las características del rostro fue necesario incluir toda la mascara

(matriz donde se detecta un rostro) y reescalarla también a un tamaño de 80x60. Finalmente, para incluir la información de los láseres se construyó un mapa local transformando los puntos de láser a una matriz de 80x60 para poderla anexar a las dos matrices de las entradas anteriores. Con esto en mente, en la figura 6 se puede visualizar un ejemplo de las entradas con las cuales se construye el estado.

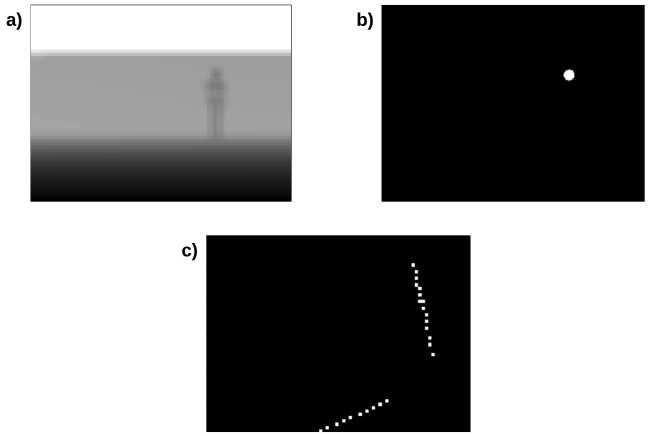

Figura 6: Entradas que conforman el estado: **a)** Imagen de profundidad **b)** máscara que simula la detección del rostro **c)** mapa local construido con la información del láser.

IV-B2. Acciones: El espacio de acciones A está definido por un total de 5 acciones discretas:

- a_0: Movimiento en x de 0,3 metros
- a_1: Movimiento en y de 0,3 metros
- a_2: Movimiento en $-y$ de 0,3 metros
- a_3: Movimiento rotacional en θ de $\pi/3$ rad.
- a_4: Movimiento rotacional en $-\theta$ de $\pi/3$ rad.

Se definen acciones de traslación de 0,3 metros con base en el diámetro aproximado del *footprint* del robot, así los movimientos resultan en pequeños pasos. De igual forma, se excluye la acción de movimiento en $-x$ (hacia atrás) dado que el robot solo incluye un sensor capaz de recopilar información en esta dirección (un sonar) y este no se incluye dentro de los estados. De esta manera, se puede evitar una navegación a ciegas.

IV-B3. Recompensa: Con el objetivo de recompensar que el robot se acerque a las personas y penalizar cuando el robot se choca, se acerca demasiado a los obstáculos o no busca de forma activa a las personas, se definió la siguiente función de recompensa.

$$r_t = R_{goal} + R_{sub-goal} + R_{laser} + R_{end} \tag{8}$$

Esta se construye a partir de cuatro componentes las cuales se describen a continuación:

$$R_{goal} = \begin{cases} +1 - 0,01 * Steps & \text{si llega al objetivo}^{*} \\ 0 & \text{en cualquier otro caso} \end{cases}$$

$$R_{sub-goal} = \begin{cases} +0{,}01 & \text{si se acerca/centra el objetivo}^{**} \\ -0{,}02 & \text{si se aleja/descentra el objetivo}^{**} \\ 0 & \text{en cualquier otro caso} \end{cases}$$

$$R_{laser} = \begin{cases} -0,1(0,5 - min(laser)) & \text{si min(laser)} < 0,5 \\ 0 & \text{en cualquier otro caso} \end{cases}$$

$$R_{end} = \begin{cases} -0,75 & \text{si se estrella con un obstáculo}^{***} \\ -0,75 & \text{si llega a 75 iteraciones}^{***} \\ -0,75 & \text{si llega a 35 iteraciones sin detectar rostro}^{***} \\ 0 & \text{en cualquier otro caso} \end{cases}$$

* Cuando el agente detecta un rostro a menos de 1.5mts en el tercio central de la imagen. **Este evento termina el episodio.**

** Cuando el recuadro del rostro detectado aumenta/disminuye en tamaño o cuando se acerca/aleja del centro de la imagen (se considera un umbral para recompensar o penalizar únicamente variaciones significativas).

*** **Estos eventos terminan el episodio.**

En este caso, el problema de navegación se modela como una tarea episódica, por lo que esta termina en uno de los siguientes tres casos: Cuando el robot llega al objetivo (note que no basta con que el robot esté a menos de 1,5 metros de la persona sino que debe estar detectando su rotstro), cuando el robot se choca contra algún obstáculo (pared o persona) o cuando se alcanza un número determinado de iteraciones, ya sea sin detectar un rostro o sin llegar a un objetivo.

V. SOLUCIÓN BASADA EN DRL

Para resolver el problema planteado anteriormente se realiza una implementación propia del algoritmo de *Deep Q-Networks (DQN)* [13] y sus variantes *Double DQN* [1] y *Clipped Double Q-Learning* [2]. El código desarrollado en Python para estas implementaciones lo puede encontrar en el siguiente repositorio: https://gitlab.com/Cesard97/rl_pepper_navigation. Al realizar esta implementación desde ceros se tuvo la posibilidad de controlar el manejo de los datos, definir una arquitectura propia y realizar unos ajustes específicos para el problema, estos se discuten a continuación:

V-A. Arquitectura del modelo (Deep Q-Network)

Con el objetivo de aprovechar al máximo los sensores del robot *Pepper* y tener una mayor observabilidad del entorno se utiliza una arquitectura con entrada multimodal para estimar la función Q, esta se presenta en la figura 7. Para esto se utilizó como base la arquitectura propuesta en [6] y de forma experimental se fueron ajustando algunos de sus parámetros.

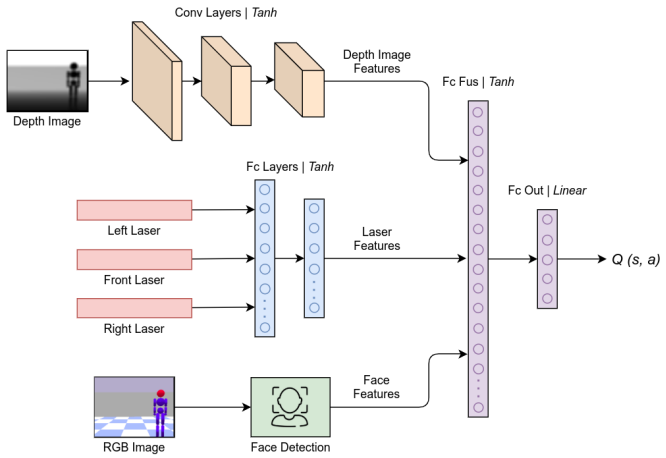

Figura 7: Arquitectura del modelo utilizada en la implementación propia

Esta se compone de tres etapas de extracción de características: Una etapa convolucional para la entrada de la imagen de profundidad, con capas de 8 filtros 8x8, 16 filtros 4x4 y 32 filtros 2x2; una etapa *Fully Connected* para la entrada de Lasers, con dos capas de 32 y 16 neuronas en las capas ocultas y un módulo de detección de rostro (en el caso de simulación este corresponde a la máscara de color) para la entrada de la imagen RGB. Estas tres etapas se unen en una red *Fully Connected* de 150 neuronas y a la salida se tienen 5 neuronas para estimar los valores Q asociados a cada una de las acciones.

V-B. Primitive prioritize Replay Buffer

En los entrenamientos se encontró que uno de los inconvenientes más recurrentes específicos al problema de este proyecto era la gran cantidad de pasos en los que el agente se encontraba ”perdido”, es decir cuando no detectaba un rostro. Por esta razón se decidió aplicar una especie de *Prioritize Experience Replay* simple, con el objetivo de dar prioridad a los estados en los que el robot detectaba un rostro y por consiguiente disminuir la prioridad de los estados en los que se encontraba ”perdido”.

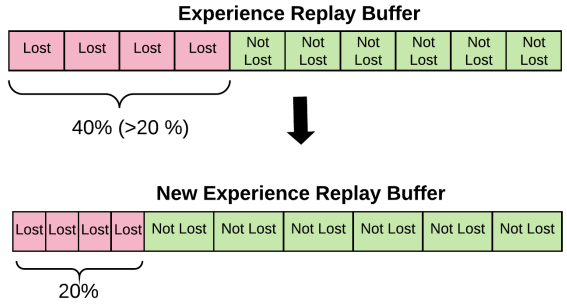

Figura 8: Implementación propia de una versión simple del *Prioritize Experience Replay Buffer*

Para esto, antes de seleccionar el *mini-batch*, se verificaba la proporción de estados en los que el agente no detectaba rostro y si dicha proporción era mayor a una probabilidad establecida como parámetro se reajustaban las probabilidades de selección de cada uno de los estados (véase figura 8). Aumentando la probabilidad de los estados cuando no estaba perdido y disminuyendo la probabilidad de cuando lo estaba al valor ingresado por parámetro (ej.20 %).

V-C. Parámetros

Adicionalmente, fue necesario ajustar varios de los parámetros con los cuales se realizó el entrenamiento. En la tabla I se presentan los parámetros finales utilizados. Para esto se realizaron pruebas variando: Tasa de aprendizaje (α) desde 0.00001 hasta 0.1, tasa de descuento (γ) entre 0,9 y 0,99 y tasa de exploración (ϵ). Para este último se consideró no solo su valor sino también su decaimiento a lo largo de los episodios (variar la exploración y explotación a largo del entrenamiento). Para esto se hicieron pruebas con decaimiento de ϵ lineal como en [9], exponencial como en [10] y asintótico. Finalmente se decidió utilizar el siguiente ϵ para todos los entrenamientos:

$$\epsilon = 0{,}05 + 0{,}95e^{-\frac{\text{Episode}}{0{,}2(\text{Exploration Episodes})}} \tag{9}$$

Tabla I: Hiperparámetros utilizados para el entrenamiento en la implementación propia

Parámetro	Valor
Tasa de descuento (γ)	0,99
Tasa de aprendizaje (α)	0,0005
Tasa de transferencia (τ)	0,01
Tasa de exploración (ϵ)	Eq. (9)
Episodios de exploración	200
Tamaño del Expirience Replay	5000
Memoria mínima	1000
Tamaño del Mini-Batch	32
Max. iteraciones por episodio	80

Vale la pena mencionar que se exploró la posibilidad de utilizar el algoritmo de DQN de la librería de aprendizaje por refuerzo *Stable Baselines* [14]. Adicional a la diferencia en el manejo de los datos de entrada, mencionada en la sección anterior, este algoritmo presentaba algunas discrepancias con la implementación propia: Utiliza modelos de red neuronal fijos, incorpora el *prioritize experience replay* propuesto en [24], utiliza una tasa de decaimiento para la exploración distinta, etc. Estas diferencias, sumadas con la diferencia en las entradas, causo que los resultados con esta implementación no tuvieran un buen desempeño, al menos con los parámetros base. Sin embargo, por restricciones de tiempo y poder computacional (dado el tiempo que tardaban las simulaciones) no fue posible hacer el ajuste adecuado de los parámetros por lo que se descartó esta solución, al menos para este proyecto.

VI. RESULTADOS

En esta sección se presentan los resultados de los entrenamientos realizados con los distintos algoritmos propuestos. Los entrenamientos se realizaron sobre el entorno sin obstáculos y se evaluaron sobre este mismo pero en condiciones predefinidas. Posteriormente se evaluó el desempeño de los mejores modelos en el entorno con paredes internas. Vale la pena aclarar que todas estas pruebas de se realizaron con una política $\epsilon - greedy$ con ϵ igual a 0,01, lo que resulta en una acción aleatoria cada 100 pasos.

VI-A. DQN

En las pruebas realizadas con el algoritmo de *DQN* no se evidenció un buen desempeño, ni una tendencia clara de aprendizaje después de 500 episodios. Esto se puede ver en la figura 9, en donde se muestra la evolución de la recompensa promedio, tomada cada 50 episodios, durante el entrenamiento. En este entrenamiento, la recompensa promedio se mantuvo casi siempre entre -0,7 y -1, recompensa mínima con la que se podía terminar los episodios. Si bien logro llegar un total 78 veces al objetivo (alrededor del 15 % de los episodios del entrenamiento) esto no parece ser resultado de una política destacable o deseada. Esto se puede observar de mejor forma en el video adjunto al final del *abstract* de este documento.

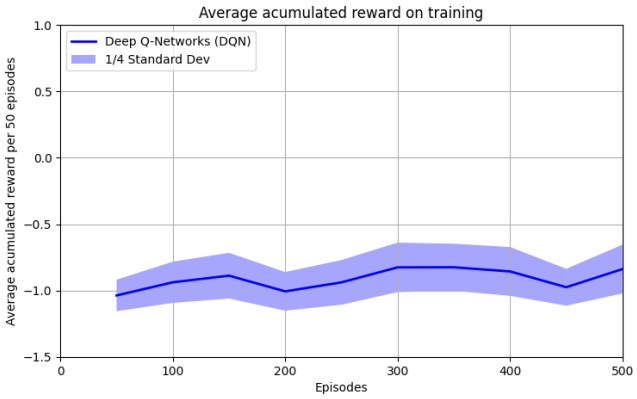

Figura 9: Recompensa acumulada promedio cada 50 episodios para el algoritmo de DQN (implementación propia) durante el entrenamiento

La causa de este pobre desempeño se especula está en el *maximization bias* que tiene el algoritmo, lo que causa una sobre-estimación de los valores Q. Esto se puede observar claramente en la figura 10 donde los valores Q, estimados por este algoritmo, alcanzan hasta 5 veces el valor de la recompensa máxima (1) y la mayor parte del tiempo están por encima de este valor. Lo que sugiere que no se hace una correcta estimación del valor de la recompensa acumulada para el estado observado ni las acciones tomadas, lo que resulta en una política con un pobre desempeño. Por esta razón se procedió a implementar algoritmos que evitaran la sobre-estimación de los valores Q, obteniendo mejores resultados.

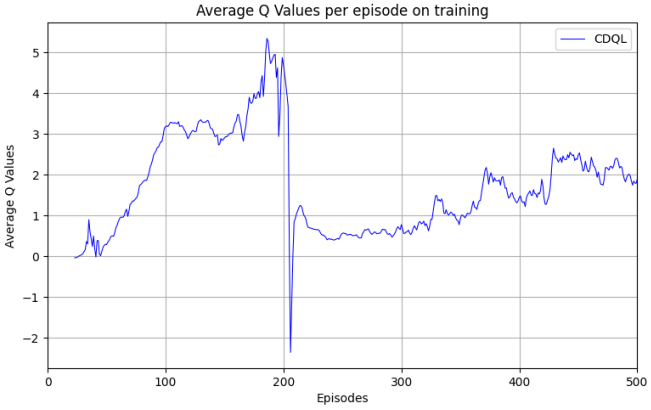

Figura 10: Valores Q estimados por el algoritmo de DQN (implementación propia) para 500 episodios de entrenamiento.

VI-B. Double DQN & Clipped Double Q-Learning

En este caso los algoritmos de *Double DQN* (DDQN) y *Clipped Double Q-Learning* (CDQL) presentaron desempeños mucho mejores y se puede evidenciar un aprendizaje a partir de la recompensa acumulada en el entrenamiento. En la figura 11 se observa la evolución de la recompensa para el entrenamiento realizado con estos dos algoritmos. En ambos casos se logra evidenciar un progreso en cuanto a su recompensa acumulada, con una ventaja de alrededor de 0,5 por parte del algoritmo de CDQL. Esta diferencia se refleja en las políticas finales que se obtuvieron con cada una de las pruebas, estas se discuten mejor en las subsecciones siguientes.

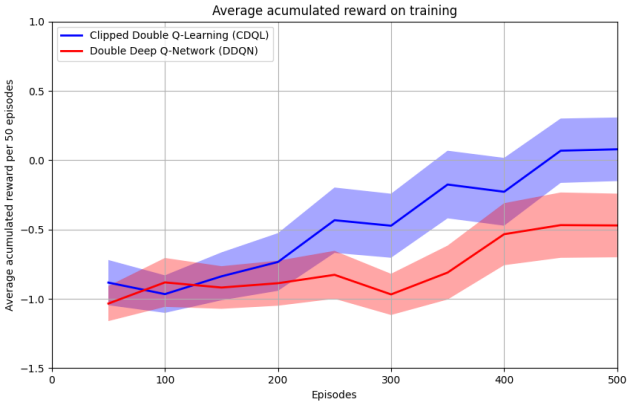

Figura 11: Recompensa acumulada promedio cada 50 episodios para algoritmos de DDQN y CDQL (implementación propia) durante el entrenamiento.

Adicionalmente, en la figura 12 se puede observar que los valores Q estimados por estos algoritmos, no solo son significativamente menores a los estimados por DQN, sino que se encuentran dentro del rango de esperado para la recompensa (-1 y 1). De igual forma, vale la pena destacar que el algoritmo de CDQL estimó valores Q menores que los que estimó DDQN, incluso percibiendo una recompensa más alta. Lo que sugiere que la modificación en la estimación de este valor (minimizar el valor Q de dos modelos distintos) es útil para reducir el *maximization bias*.

Figura 12: Valores Q promedio estimados por los algoritmos de implementación propia a lo largo del entrenamiento.

VI-C. Desempeño de las mejores políticas en simulación

Para validar el desempeño de las políticas obtenidas con los algoritmos de DDQN y CDQL, se decidió primero evaluar el modelo en el mismo escenario de entrenamiento (espacio libre, sin obstáculos) y posteriormente evaluarlo en un entorno con paredes intermedias (véase figura 5). Sin embargo, para tener una mayor certeza del desempeño en el mapa de entrenamiento se utilizó un único objetivo, es decir una única persona en el mapa, y se inicio con esta persona en distintos lugares del mapa siguiendo un patrón. Asimismo, el agente (robot Pepper) inició todos los episodios siempre en el centro del entorno. Con esto es posible determinar en que lugares le costó más al agente llegar al único objetivo dentro del cuarto. Los resultados de estas pruebas se pueden ver en las figuras 13 y 14 (estas se presentan en conjunto con el rango de visión del robot).

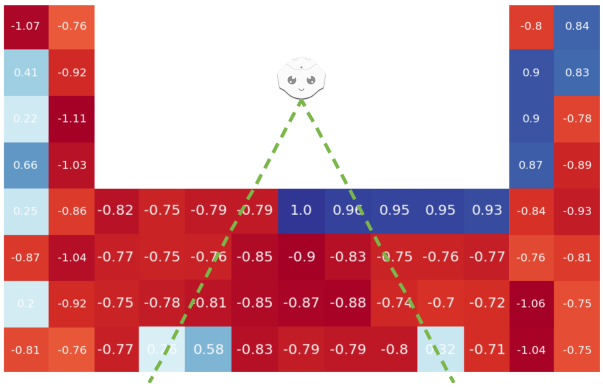

Figura 13: Recompensa acumulada por zona de inicio del objetivo para el modelo entrenado con DDQN

La política resultante del algoritmo de Double DQN, resultó no tener un buen desempeño en el escenario de pruebas, alcanzando el objetivo únicamente el 26 % de los episodios. Esto se puede entender mejor al observar el comportamiento del agente, (véase enlace adjunto al final del *abstract* de este documento). La política resultante de este algoritmo lleva al agente siempre hacia la izquierda hasta encontrar una pared, la cual sigue, la mayoría de las veces sin estrellarse y girando en las esquinas. Si bien, esto le permite recorrer el entorno de forma relativamente segura, omite pasar por las zonas centrales y no le permite encontrar a las personas que se encuentran en este lugar, únicamente explora los alrededores de la habitación. Es por esta razón que la mayoría de recompensas en la figura 13 son negativas y únicamente en algunos bordes es que logra el agente encontrar a las personas (recompensa positiva).

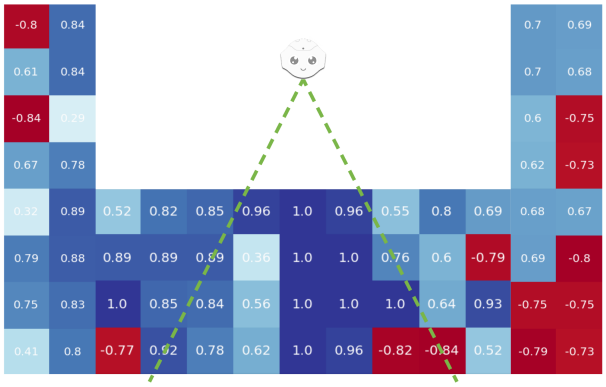

Figura 14: Recompensa acumulada por zona de inicio del objetivo para el modelo entrenado con CDQL

Por su parte, la política resultante del algoritmo de CDQL presentó un desempeño mucho mejor en el escenario de pruebas, alcanzando el objetivo el 81 % de las veces. Adicionalmente, en los cuatro episodios donde no logró llegar al objetivo no se estrelló, sino que el episodio termino luego del límite máximo de iteraciones (75). Al analizar el comportamiento se puede observar que en este caso el agente prioriza el movimiento hacia adelante hasta que encuentra alguna pared, momento en el que gira y acomoda nuevamente su orientación buscando el centro de la habitación, lo que le permite encontrar de mejor forma a las personas.

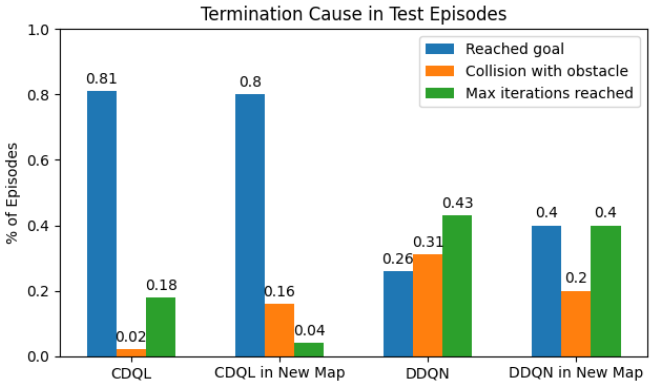

Figura 15: Resumen de resultados de las pruebas realizadas a los modelos de CDQL y DDQN en los dos mapas propuestos

Ahora bien, al momento de evaluar el desempeño de los modelos en un entorno distinto, los resultados fueron similares (véase figura 15). En este caso, el comportamiento fue bastante

similar al presentado en el entorno sin obstáculos. No obstante, el modelo entrenado con CDQL en esta ocasión se estrelló al tratar de acercarse a una persona que detectaba al otro lado de la pared. Por su parte, el modelo entrenado con DDQN mejoró su desempeño al lograr alcanzar el objetivo más veces dada la disposición de las personas y las paredes en el entorno.

VII. Discusión y Conclusiones

En este proyecto de grado se logró desarrollar un sistema de navegación *learning-based* para el robot *Pepper* utilizando aprendizaje por refuerzo profundo. Este le permitió al robot navegar hasta las personas que detectaba con su cámara RGB en más del 80 % de los casos evaluados en simulación. Adicionalmente, se implementaron tres algoritmos distintos de *Q-learning*, sobre los cuales se pudo evidenciar el problema con respecto al *maximization bias* y el efecto que tiene en el desempeño del agente.

De los algoritmos implementados en este trabajo, se pudo evidenciar que las mejoras que proponen *Double DQN* y *Clipped Double Q-Learning*, no solo reducen la sobreestimación de los valores Q, sino que mejoran el desempeño de la política resultante. Esto se puede evidenciar en la evolución de la recompensa acumulada con el paso del tiempo y la reducción en la volatilidad de los valores Q estimados. Adicionalmente, se observa que la propuesta de CDQL estima valores Q ligeramente menores que los de DDQN, incluso percibiendo una recompensa mayor, esto resultando también en un mejor desempeño. No obstante, no se puede asegurar de forma categórica que el algoritmo de CDQL sea mejor que el de DDQN, pues existen muchas otras variables que afectan el resultado del entrenamiento.

Sin embargo, una explicación intuitiva de porque en este caso CDQL presentó un mejor desempeño, se puede dar a partir de los comportamientos resultantes de ambos modelos. El primero, entrenado con DDQN, resultó en una navegación que priorizaba el movimiento a la izquierda y recorrer el mapa siguiendo las paredes. Por su parte, el segundo modelo, entrenado con CDQL, priorizaba el movimiento hacia adelante hasta encontrar una pared, momento en el que giraba buscando alejarse de esta. Asimismo, cuando detectaba una persona buscaba alinearse y activamente acercarse a esta. Con esto en mente, se podría pensar que la sobre estimación de valores Q causan que el modelo se conforme con una búsqueda relativamente pasiva alrededor del entorno. Mientras que cuando se subestiman los valores Q, el modelo resultante es más pesimista ante su estado y busca de una forma más activa llegar a las personas. Esto concuerda con lo estipulado por Fujimoto et al. en [2] en donde sugiere que esta subestimación de valores podría resultar en un comportamiento más deseado.

Adicionalmente, vale la pena destacar que los algoritmos implementados lograron un entrenamiento relativamente rápido, en aproximadamente 500 episodios, lo cual representa alrededor de 25.000 pasos de simulación. Dado el tiempo que tomaban las simulaciones, el cual era alto en comparación con otros *frameworks*, en este proyecto se buscó hacer el entrenamiento lo más eficiente posible. Razón por la cual se implementó el *primitive experience replay* y se construyó el escenario de entrenamiento con varios objetivos (personas), tratando de facilitar y acelerar el aprendizaje. No obstante, este poco tiempo de entrenamiento puedo ser la causa de que algunas de las pruebas no funcionaran, como fue el caso de la implementación con la librería de *Stable Baselines*. El hecho de que esta implementación utilizara entradas distintas (explicadas en la sección de IV-B) pudo significar que el aprendizaje requiriera de más tiempo. Esto pues el modelo debe aprender mucho más, especialmente de la máscara y del mapa local, a diferencia de la implementación propia.

A partir de estos resultados y como trabajo futuro, se podrían realizar pruebas y entrenamientos sobre entornos más complejos que requieran de una evasión de obstáculos robusta. Asimismo, se podría esperar que una sistema de este estilo llegue a implementarse en el robot *Pepper* en físico. No obstante, esto requiere de una etapa de calibración entre los datos reales y los simulados y un pequeño desarrollo en el *framework* de ROS para ser posible. Finalmente, se podría mejorar la propuesta hecha en este trabajo, explorando la implementación de otros algoritmos de aprendizaje por refuerzo, como los *Actor-Critic*: DDPG, A2C, TD3, etc. Con los cuales se esperaría alcanzar un mejor desempeño e incluso acciones continuas que permitan un movimiento más fluido del agente.

Referencias

[1] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, p. 2094–2100, AAAI Press, 2016.

[2] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *ArXiv*, vol. abs/1802.09477, 2018.

[3] M. Busy and M. Caniot, "qibullet, a bullet-based simulator for the pepper and nao robots," 09 2019.

[4] S. Nurmaini and B. Tutuko, "Intelligent robotics navigation system: Problems, methods, and algorithm," *International Journal of Electrical and Computer Engineering*, vol. 7, 12 2017.

[5] R. Groot, "Autonomous exploration and navigation with the pepper robot," Master's thesis, 2018.

[6] F. Leiva, K. Lobos-Tsunekawa, and J. Ruiz-del Solar, "Collision avoidance for indoor service robots through multimodal deep reinforcement learning," in *RoboCup 2019: Robot World Cup XXIII* (S. Chalup, T. Niemueller, J. Suthakorn, and M.-A. Williams, eds.), (Cham), pp. 140–153, Springer International Publishing, 2019.

[7] C. Gómez, M. Mattamala, T. Resink, and J. Ruiz-del Solar, "Visual slam-based localization and navigation for service robots: The pepper case," 06 2018.

[8] G. Bresson, Z. Alsayed, L. Yu, and S. Glaser, "Simultaneous localization and mapping: A survey of current trends in autonomous driving," *IEEE Transactions on Intelligent Vehicles*, vol. 2, no. 3, pp. 194–220, 2017.

[9] X. Ruan, D. Ren, X. Zhu, and J. Huang, "Mobile robot navigation based on deep reinforcement learning," in *2019 Chinese Control And Decision Conference (CCDC)*, pp. 6174–6178, 2019.

[10] X. Xue, Z. Li, D. Zhang, and Y. Yan, "A deep reinforcement learning method for mobile robot collision avoidance based on double dqn," in *2019 IEEE 28th International Symposium on Industrial Electronics (ISIE)*, pp. 2131–2136, 2019.

[11] J. Kulhánek, E. Derner, T. de Bruin, and R. Babuška, "Vision-based navigation using deep reinforcement learning," in *2019 European Conference on Mobile Robots (ECMR)*, pp. 1–8, 2019.

[12] A. Zuluaga, F. E. Lozano, C. Higuera, and L. F. Giraldo, "Atención al cliente por medio de reconocimiento facial," 2019.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.
[14] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." https://github.com/hill-a/stable-baselines, 2018.
[15] R. Smith and P. Cheeseman, "On the representation and estimation of spatial uncertainty," *The International Journal of Robotics Research*, vol. 5, 02 1987.
[16] J. J. Leonard and H. F. Durrant-Whyte, "Simultaneous map building and localization for an autonomous mobile robot," in *Proceedings IROS '91:IEEE/RSJ International Workshop on Intelligent Robots and Systems '91*, pp. 1442–1447 vol.3, 1991.
[17] H. Tawade, J. Trivedi, and G. Haldankar, "An empirical intelligent agent for robot navigation problem using supervised learning," in *2016 IEEE 1st International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES)*, pp. 1–5, 2016.
[18] D. Barnes, W. Maddern, and I. Posner, "Find your own way: Weakly-supervised segmentation of path proposals for urban autonomy," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 203–210, 2017.
[19] G. Kahn, A. Villaflor, B. Ding, P. Abbeel, and S. Levine, "Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5129–5136, 2018.
[20] A. H. Qureshi, Y. Nakamura, Y. Yoshikawa, and H. Ishiguro, "Robot gains social intelligence through multimodal deep reinforcement learning," in *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, pp. 745–751, 2016.
[21] A.-L. Vollmer and N. J. Hemion, "A user study on robot skill learning without a cost function: Optimization of dynamic movement primitives via naive user feedback," *Frontiers in Robotics and AI*, vol. 5, p. 77, 2018.
[22] C. J. C. H. Watkins, *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.
[23] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.
[24] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *CoRR*, vol. abs/1511.05952, 2016.
[25] H. V. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems 23* (J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds.), pp. 2613–2621, Curran Associates, Inc., 2010.
[26] Aldebaran, "Pepper documentation." http://doc.aldebaran.com/2-4/home_pepper.html, 2019. Accessed: 2020-07-10.