Universidad Zaragoza
1542

Trabajo Fin de Máster

Estrategias de Deep Learning en SLAM Activo

Deep Learning Strategies in Active SLAM

Autor

Julio Alberto Placed Perales

Director

José Ángel Castellanos Gómez

Escuela de Ingeniería y Arquitectura

2019

Abstract

Active SLAM (Simultanous Localisation and Mapping) refers to the problem of controlling the movement of a robot while performing SLAM, so as to minimize the map representation and robot localisation's uncertainty. This problem has traditionally been solved by filtering methods or other frameworks that involve Markov decision processes or reinforcement learning algorithms. In those solutions, it is necessary to (i) identify the possible actions to take, (ii) compute the expected future value of each one (e.g. via utility functions) and (iii) select the optimal action.

In this Master's Thesis, we analyse the resolution of the problem by using deep neural networks, a currently booming field where supervised learning is the preferred learning form par excellence, outshinning other methods in the literature. Active SLAM nature, however, makes it necessary to use a different machine learning method: deep reinforcement learning. We analyse the potential and limitations of this framework, usually executed on simple simulation environments in which also the difference between exploration and navigation and generalisation issues are frequently ignored.

Several reinforcement and deep reinforcement learning approaches based on Q-learning have been implemented on top of Gazebo simulator. Both learning processes and the agent's ability to generalise are deeply studied, achieving trained agents capable of navigating on unseen environments. Moreover, the inclusion of covariance matrix metrics in the algorithm's reward function is proposed, achieving a gradual entropy decrease during exploration and encouraging much more optimal actions in terms of uncertainty reduction.

Resumen

El SLAM (*Simultanous Localisation and Mapping*) activo hace referencia al problema de controlar el movimiento de un robot que está realizando SLAM, de forma que se minimice la incertidumbre del mapa creado y de su localización. Tradicionalmente ha sido resuelto mediante filtros u otras aproximaciones que involucran procesos de decisión de Markov o algoritmos de aprendizaje por refuerzo. En éstos, es necesario (i) identificar las posibles acciones, (ii) calcular el valor futuro esperado de cada una de ellas (e.g. mediante funciones de utilidad) y (iii) ejecutar la acción óptima.

En este Trabajo Fin de Máster se analiza la resolución del problema mediante redes neuronales profundas, un campo de gran auge en la actualidad donde el aprendizaje por excelencia es el supervisado, que atrae la mayoría de investigaciones y aplicaciones de la literatura. La naturaleza del problema abordado, sin embargo, hace necesario el uso de otra forma de aprendizaje automático: el aprendizaje por refuerzo profundo. Se ha analizado el potencial y las limitaciones de este marco de trabajo, empleado normalmente en entornos de simulación sencillos, donde la diferencia entre exploración y navegación y el problema de generalización (clave en el SLAM activo, puesto que la información *a priori* del entorno es nula) son habitualmente obviados.

Se han implementado distintas aproximaciones de aprendizaje por refuerzo y refuerzo profundo basadas en *Q-learning* sobre el entorno de simulación Gazebo. Ambos aprendizajes y su capacidad de generalización a escenarios desconocidos se estudian en profundidad, consiguiendo que agentes entrenados naveguen por entornos totalmente desconocidos. Además, se propone la inclusión de una métrica de la matriz de covarianza en la función de recompensa, consiguiendo una reducción de entropía paulatina durante la exploración y favoreciendo acciones mucho más óptimas en términos de reducción de la incertidumbre.

A mi familia.

Índice

Capítulo 1

Introducción

En este capítulo se presentan la motivación, el alcance, los objetivos contemplados, la metodología seguida y la estructura de este Trabajo Fin de Máster.

1.1. Motivación, Alcance y Objetivos

En los últimos años, el aprendizaje automático mediante redes neuronales cada vez más profundas ha tenido un crecimiento asombroso y ha comenzado a aplicarse en la mayoría de áreas de investigación, incluida la robótica. Ejemplos de ello son la detección de lazos en SLAM (*Simultaneous Localisation and Mapping*), la eliminación de elementos dinámicos de imágenes, o incluso soluciones *end-to-end* de SLAM monocular basado en redes convolucionales. En medio de este auge de las redes profundas entrenadas mediante aprendizaje supervisado, cabe preguntarse si podrían ser empleadas en problemas menos preparados para ello, como la toma de decisiones, donde otros tipos de aprendizaje menos estudiados toman protagonismo y en donde la capacidad de generalización es imprescindible.

Este Trabajo Fin de Máster surge para analizar el potencial y las limitaciones de las redes profundas en un área de la robótica que ha sido estudiada durante décadas: el SLAM activo, donde los conocimientos adquiridos por el agente han de extrapolarse a entornos distintos al de entrenamiento cuyo conocimiento *a priori* es nulo, donde debe aprenderse mediante la interacción con el propio entorno y donde además el estado del agente puede no ser completamente observable. Estas premisas hacen que la resolución del problema no sea trivial y que el aprendizaje supervisado, el área más estudiada del aprendizaje automático, no tenga cabida.

En primer lugar, se pretende entender los conceptos de SLAM activo y aprendizaje profundo, así como analizar el estado del arte de las técnicas de aprendizaje profundo aplicadas al problema de exploración robótica hasta el momento. Este análisis permitirá explorar el potencial y las limitaciones de

las redes neuronales en este contexto y proponer algunas líneas de trabajo e implementaciones. Pese a que el trabajo contiene un amplio capítulo de implementación de algoritmos de aprendizaje por refuerzo profundo ejecutados sobre Gazebo, su núcleo principal ha sido el análisis de las estrategias empleadas en la actualidad.

1.2. Estructura

El documento restante está estructurado de la siguiente manera. El capítulo 2 contiene una breve introducción al SLAM activo y las funciones de utilidad, las redes neuronales artificiales y el aprendizaje profundo; conceptos clave que se usarán en adelante. En el tercer y cuarto capítulos se desarrolla la idea principal de este trabajo, analizar el uso de *Deep Learning* en SLAM activo. El capítulo 3 expone una amplia revisión bibliográfica sobre ésta, mientras que en el 4 se implementan y evalúan distintas soluciones. Finalmente, se presentan las conclusiones del trabajo realizado e ideas de trabajo futuro en el capítulo 5.

Además, se incluyen los siguientes anexos a la memoria:

- Anexo A: descripción de dos de los algoritmos básicos de *Reinforcement Learning*: *Q-learning* y *Policy Gradient*.
- Anexo B: descripción y detalles de algoritmos de *Deep Reinforcement Learning*: *Deep Q-Networks*.

Capítulo 2

Antecedentes

En este capítulo introductorio, se presentan teóricamente los dos conceptos sobre los que trata este Trabajo Fin de Máster: el SLAM activo y las redes neuronales profundas. El capítulo está estructurado como sigue:

- En primer lugar se presentan los conceptos de SLAM y SLAM activo, incidiendo de manera especial en los procesos de decisión de Markov y en las funciones de utilidad (sección 2.1).
- Tras esto, se presenta la teoría sobre redes neuronales artificiales, así como una breve reseña de su evolución. Se explican en mayor profundidad las redes convolucionales (sección 2.2).
- Finalmente se introducen los conceptos de de *backpropagation*, aprendizaje profundo y los tipos de aprendizaje más comunes (sección 2.3).

2.1. SLAM Activo

El mapeo y localización simultáneos, SLAM (*Simultaneous Localisation and Mapping*), consiste en la determinación de la posición de un robot móvil en un entorno desconocido a la vez que se crea un mapa de este entorno de forma incremental o recurrente.

Históricamente, el problema se ha resuelto empleando la odometría visual (*Visual Odometry*, VO) en el entorno de, habitualmente, un filtro extendido de Kalman (*Extended Kalman Filter*, EKF) en el que ambos problemas de localización y mapeado se resuelven conjuntamente en un proceso iterativo que consta de: (i) extracción de características del entorno (*features*), (ii) asociación de datos entre mediciones, (iii) estimación y actualización del estado y (iv) actualización de características (Thrun et al., 2005; Piniés-Rodríguez, 2009; Viñal-Pons, 2012). Otras aproximaciones también basadas en filtros han proliferado desde entonces, como los filtros de partículas, los filtros de información o *Unscented Transforms*.

De manera más reciente y en contraste a las aproximaciones anteriores, se ha comenzado a usar no sólo la información visual sino también la inercial en un marco de optimización no lineal, integrando el problema de VIO (*Visual-Inertial Odometry*) en SLAM. Ejemplos de ello son (Burusa, 2017) o (Mur-Artal & Tardós, 2017b), donde se propone la integración de la información visual-inercial bajo el marco de ORB-SLAM2, formulado por los mismos autores en (Mur-Artal et al., 2015; Mur-Artal & Tardós, 2017a). El aumento de la capacidad computacional ha favorecido el crecimiento de estas aproximaciones basadas en la minimización de una función que incluye el error fotométrico o el de reproyección y, habitualmente, una métrica de la información inercial (i.e. *tightly-coupled methods*). Ver (Leutenegger et al., 2015; Concha et al., 2016; Usenko et al., 2016; Forster et al., 2017).

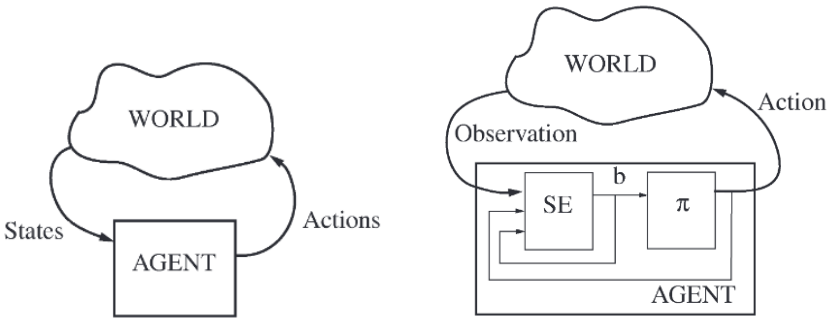
La exploración robótica fue definida por Thrun et al., (2005, cap. 17) como el problema de controlar un robot de forma que su conocimiento sobre el entorno sea óptimo. Para ello, parece lógico pensar que es necesaria una buena estimación de la pose del robot y, por ende, una reconstrucción certera del mapa del entorno. El SLAM activo (*active* SLAM), a veces denominado SPLAM (*Simultaneous Planning, Localisation and Mapping*), resuelve el problema de la exploración robótica empleando técnicas de SLAM. De acuerdo a Carrillo-Lindado (2014, cap. 3), puede definirse como:

> *"... the problem of controlling the movements of a robot performing SLAM so as to maximize the accuracy of its map representation and localization."*

2.1.1. Procesos de decisión de Markov

Por las características del problema de toma de decisiones, el SLAM activo está englobado dentro del marco matemático de los procesos de decisión de Markov parcialmente observables (*Partially Observable Markov Decision Process*, POMDP); que formalmente están definidos por la 7-tupla $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, \gamma)$, donde $\mathcal{S}$ es el conjunto finito de estados, $\mathcal{A}$ el de acciones, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto \Pi(\mathcal{S})$ el conjunto de probabilidades condicionales entre estados (i.e. la probabilidad de terminar en el estado s' tras haber ejecutado la acción a en el estado s), $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ la función de recompensa, Ω el conjunto finito de observaciones y $\mathcal{O} : \mathcal{S} \times \mathcal{A} \mapsto \Pi(\Omega)$ sus probabilidades condicionales (i.e. la probabilidad de realizar la observación o tras haber ejecutado la acción a y haber terminado en el estado s), y $\gamma \in [0, 1]$ el factor de descuento que permite trabajar siempre en un horizonte temporal finito. El objetivo del agente es actuar de forma que se maximice una métrica de la recompensa a largo plazo. La función de recompensa empleada habitualmente es la recompensa futura descontada esperada:

$$R_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{t=0}^{\infty} \gamma^t r_{t+k+1} = \sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \qquad (2.1)$$



(a) MDP (b) POMDP

Figura 2.1: Estructura de (a) un MDP, donde se modela la interacción entre un agente y el entorno; y (b) un POMDP. De (Kaelbling et al., 1998).

Este *framework* no es más que un proceso de decisión de Markov (MDP) en el que el agente es incapaz de observar el estado actual – o parte de él –, y que ha de tomar decisiones con incertidumbre en el estado real. Mediante la interacción con el entorno, el agente puede actualizar su estimación (*belief*) del estado real actualizando la distribución de probabilidad del estado actual. En la figura 2.1 se pueden observar las diferencias entre un MDP y su homólogo parcialmente observable, que ha de contener un bloque de estimación (*state estimation*, SE).

La creencia o *belief* de un estado será la probabilidad de distribución sobre $\mathcal{S}$. Sea $b(s)$ la probabilidad de que $b \in \mathcal{B}$ sea el estado real $s \in \mathcal{S}$, entonces

$$0 \leq b(s) \leq 1 \tag{2.2}$$

$$\sum_{s \in \mathcal{S}} b(s) = 1 \tag{2.3}$$

Y este *belief* puede ser actualizado a partir de la estimación anterior siguiendo la teoría de probabilidad como:

$$b'(s') \doteq \mathbb{P}[s'|o, a, b] = ... = \frac{\mathcal{O}(o, a, s') \sum_{s \in \mathcal{S}} \mathcal{T}(s, a, s') b(s)}{\mathbb{P}[o|a, b]} \tag{2.4}$$

La resolución de estos problemas se verá en detalle en el capítulo 3, donde se analizarán distintas aproximaciones clásicas y las basadas en redes neuronales y se explicarán otros conceptos importantes como las funciones de valor y las reglas de comportamiento.

2.1.2. Funciones de utilidad

En términos generales, el SLAM activo consta de tres fases (Carrillo-Lindado, 2014; Rodríguez-Arévalo, 2018): (i) la identificación de las posibles localizaciones a explorar, (ii) el cálculo de la recompensa o ventaja que genera cada una de las acciones que conducen a estas localizaciones de interés, y (iii) la ejecución de la acción más ventajosa. La elección de puntos de interés (*vantage points*) puede realizarse de forma aleatoria, basándose en las localizaciones vecinas, en la frontera entre zonas conocidas y desconocidas, etc. Tras identificarlos, se debe crear un conjunto de posibles acciones que conduzcan al robot a dichos puntos, donde cada acción tendrá asociada una cuantificación de su utilidad que, en última instancia, no será más que una cuantificación de la incertidumbre del mapa y/o de la pose del robot. Finalmente, basta con escoger la acción más ventajosa mediante la optimización de la función de utilidad.

En función de como estén formuladas, las funciones de utilidad se dividen en dos grupos: las orientadas a la tarea (*task-driven*) y las orientadas a la información (*information-driven*). Las primeras también se denominan criterios de optimalidad y emplean funciones escalares de la matriz de covarianza para cuantificar la incertidumbre. Pueden verse como una cuantificación del volumen de la (hiper)elipsoide correspondiente a la matriz de covarianza. Los criterios de optimalidad más empleados son los siguientes (Carrillo-Lindado, 2014; Rodríguez-Arévalo, 2018):

- Criterio de optimalidad A: cuantifica la traza de la matriz de covarianza – i.e. la suma de sus valores propios (*A-opt*, eq. (2.5)).
- Criterio de optimalidad D: cuantifica el determinante de la matriz de covarianza – i.e. el producto de sus valores propios (*D-opt*, eq. (2.6)).
- Criterio de optimalidad E: cuantifica el mayor de los valores propios (*E-opt*, eq. (2.7)).

$$\mathrm{tr}(\Sigma) \doteq \frac{1}{n}\sum_{i=1}^{n}\lambda_i \tag{2.5}$$

$$\det(\Sigma) \doteq \exp\left(\frac{1}{n}\sum_{i=1}^{n}\log(\lambda_i)\right) \tag{2.6}$$

$$\mathrm{máx}(\lambda_i) \tag{2.7}$$

Donde $\Sigma \in \mathbb{R}^{n\times n}$ es la matriz de covarianza, λ_i son los valores propios de dicha matriz y n la dimensión del vector de estado.

Habitualmente se emplea uno de estos tres criterios, y aunque los criterios *A-opt* y *E-opt* requieren de una menor capacidad de computación, es fácil notar que el criterio *D-opt* es el único capaz de capturar la incertidumbre global, como ya expuso Kiefer (1974), puesto que actúa sobre todos los elementos de Σ.

Por otra parte, existen funciones de utilidad basadas en la Teoría de la Información, siendo la entropía es la más empleada. Sea $\mathcal{P}$ una distribución

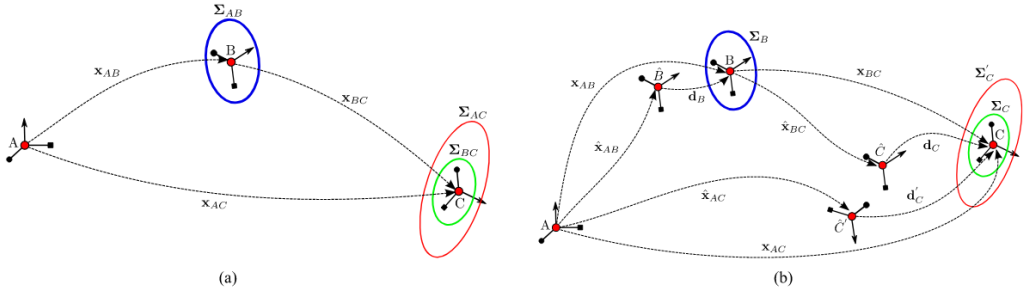


Figura 2.2: Composición de poses empleando la representación (a) absoluta y (b) diferencial. De (Rodríguez-Arévalo et al., 2018).

Gaussiana, entonces su entropía de Shannon será:

$$\mathcal{H}(\mathcal{P}) \doteq \frac{n}{2}\left(1 + \log(2\pi)\right) + \frac{1}{2}\log(\det(\Sigma)) \tag{2.8}$$

Y dado que $\mathcal{H} \propto \det(\Sigma)$, su comportamiento será similar al del criterio *D-opt*.

Ambos grupos de funciones de utilidad se basan en la cuantificación de la incertidumbre de la pose del robot, por lo que su representación es de vital importancia. Existen dos formas de representar esta incertidumbre: absoluta y diferencial. La incertidumbre absoluta asigna una función de distribución a la pose absoluta del robot respecto a una cierta referencia (x_{AB}); mientras que la incertidumbre diferencial supone una pose aproximada (estimación, $\hat{x}_{AB}$) y asigna la función de distribución al error de dicha aproximación (d_B), como se muestra en la figura 2.2. En (Rodríguez-Arévalo, 2018) se prueba analíticamente que la propiedad de monotonía se mantiene en todas las funciones de utilidad –y en la entropía de Shannon– cuando la incertidumbre es representada de forma diferencial; comportamiento de gran importancia en tanto en cuanto el sistema escogerá aquellas acciones que generen una menor incertidumbre en la estimación de la pose.

2.2. Redes Neuronales Artificiales

2.2.1. Contexto histórico

El uso de neuronas como modelos matemáticos fue propuesto por primera vez por McCulloch y Pitts (1943), empleando estas unidades como funciones lógicas (e.g. AND, OR, XOR) donde las entradas eran señales binarias. Dos décadas después, sería Rosenblatt (1961) quien, basándose en el modelo de McCulloch-Pitts y las investigaciones de Hebb (1949), acuñase el concepto de perceptrón. Se trata una neurona artificial similar a la propuesta por sus predecesores en la que cada señal de entrada está ponderada con un factor w, y la función de activación es la función signo. La red, formada por una sola capa, podía entrenarse calculando la salida a partir de una entrada dada y comparándola con la salida esperada. De esta forma, se podían ajustar las ponderaciones para que ambas coincidiesen (aprendizaje supervisado).

Este modelo fue duramente criticado en (Minsky & Papert, 1969), donde se demostró su limitación a la hora de modelar problemas linealmente no separables (e.g. función XOR). Además, demostraron que el método de aprendizaje de McCulloch no funcionaba para las redes con múltiples capas (*Multilayer Perceptron*, MLP), necesarias para modelar este tipo de problemas.

El desarrollo del campo de las redes neuronales se paralizó hasta la década de 1980, cuando hitos como el uso de las redes neuronales recurrentes de Hopfield (1982), las redes convolucionales de LeCun en 1989, o el "redescubrimiento" del algoritmo de *backpropagation* por Rumelhart et al., (1986) incitaron a numerosos científicos a continuar la investigación en este campo. Mediante este método de minimización de errores basado en el gradiente descendente, se podían entrenar redes multicapa.

Tras estos avances, la investigación se extendió también al aprendizaje no supervisado (*autoencoders*), las redes profundas (*Deep Neural Networks*, DNN), las redes convolucionales (*Convolutional Neural Networks*, CNN) o las redes de creencia profunda (*Deep Belief Networks*, DBN), entre otros.

2.2.2. Redes neuronales artificiales

Una red neuronal artificial (*Artificial Neural Network*, ANN) es una red formada por nodos (neuronas) conectados entre sí, de tal forma que la salida de unos sea la entrada de otros. Las neuronas son capaces de recibir señales de entrada, cambiar su estado interno (activación) y producir señales de salida, de igual forma que ocurre en el cerebro. El potencial de esta herramienta para modelar sistemas complejos reside en la agrupación de neuronas en distintas capas dentro de la red, de forma que la información de entrada se propaga (habitualmente hacia delante) por sucesivas transformaciones de carácter no lineal.

Considérese una ANN con una capa de entrada, una capa de salida, n capas intermedias (*hidden layers*) y un cierto número de neuronas en cada capa (*hidden units*). Puesto que la conexión entre neuronas de distintas capas transfiere la salida de una de ellas (predecesora) como entrada a la siguiente (sucesora) y cada una de estas conexiones tiene asignada una ponderación w, se tratará de un grafo dirigido ponderado.

Cuando una neurona dentro del grafo recibe señales de entrada de neuronas predecesoras, se computa la suma ponderada de todas las entradas (conocida como función de propagación o excitación). Además, para conseguir un comportamiento no lineal en la red, debe existir una segunda operación, conocida como función de activación (ver figura 2.3). Ésta actúa como una capa adicional entre capas ocultas, transformando la salida de cada neurona por una función no lineal que permite acotar la salida. Las funciones de activación más comunes son la función sigmoide, el rectificador (*Rectified Linear Unit*, ReLU) o la tangente hiperbólica, definidas respectivamente como (ver figura 2.4):

$$y(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{2.9}$$

$$y(x) = \mathrm{máx}(0, x) \tag{2.10}$$

$$y(x) = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-x}} \tag{2.11}$$

Cada capa puede estar totalmente conectada (*Fully Connected Layer*, FCL) con la capa predecesora (capa densa, i.e. en la entrada de cada neurona de la capa están consideradas las salidas de todas las neuronas de la capa anterior) o parcialmente conectada. Las ventajas en cuanto al coste computacional de conectar las capas localmente entre sí es clara, y por ello estas capas son comúnmente empleadas en aplicaciones con grandes volúmenes de información (e.g. reconocimiento de imágenes, reconocimiento de audio, procesamiento de lenguaje, clasificación de texto...). Un ejemplo de capas parcialmente conectadas son las capas convolucionales.

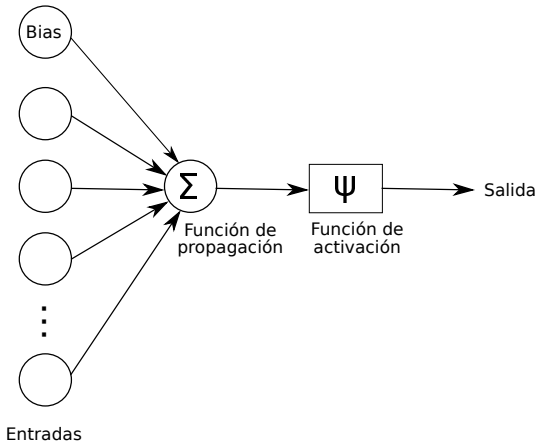


Figura 2.3: Modelo de una neurona artificial.

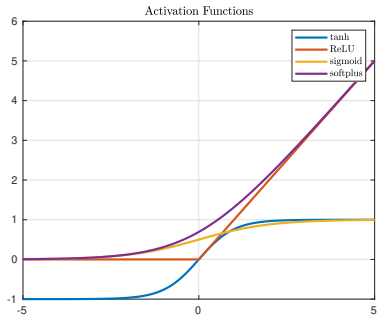


Figura 2.4: Funciones de activación más habituales: sigmoide (amarillo), tangente hiperbólica (azul), ReLU (rojo) y softmax (violeta).

Una red neuronal convolucional (*Convolutional Neural Network*, CNN) está formada por capas convolucionales apiladas. Esta arquitectura favorece que cada neurona se especialice en una región de la capa predecesora, reduciendo la necesidad computacional (e.g. sea más sensitiva a ciertas características una imagen, como líneas o esquinas). Las CNN están habitualmente formadas por combinaciones de:

(a) capas convolucionales que aplican filtros de un cierto tamaño a la entrada completa (habitualmente una imagen)

(b) capas de reducción de muestreo (*pooling*): se produce una reducción progresiva del tamaño espacial de la representación, disminuyendo así el número de parámetros y la necesidad de computación de la red (y por tanto ayudando a evitar el sobreajuste)

(c) capas densas (normalmente a la salida)

En la figura 2.5 puede verse la estructura típica de una CNN con los tres tipos de capas mencionados.

Por ejemplo, en una aplicación de procesamiento de imágenes, las capas convolucionales aplican un cierto número de filtros o *kernels* de convolución a las imágenes (e.g. 64 filtros de tamaño 5×5 píxeles). Para cada una de estas subregiones de tamaño 5×5, se genera un único valor de salida al que se aplica la función de activación para introducir no linealidades en el modelo (habitualmente ReLU). Sucesivas capas convolucionales de distinto número de filtros pueden apilarse. Los hiperparámetros de estas capas son el número de filtros (K), su dimensión, su paso (*stride*) y la cantidad de *zero-padding*. Las capas de *pooling* se sitúan periódicamente entre capas convolucionales para reducir progresivamente el tamaño necesario de representación, el número de parámetros y la computación de la red. Un ejemplo de estas capas es *max-pooling*, con el que se dividen las salidas de la capa convolucional en subconjuntos más pequeños y se selecciona únicamente el mayor valor de los existentes. Los hiperparámetros de estas capas son el tamaño de la salida y el paso. Finalmente, a la salida se sitúa habitualmente una o más capas densas para clasificar las características extraídas por las capas predecesoras, con un número de neuronas igual

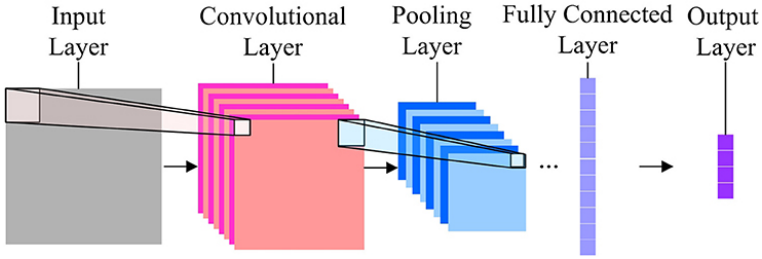
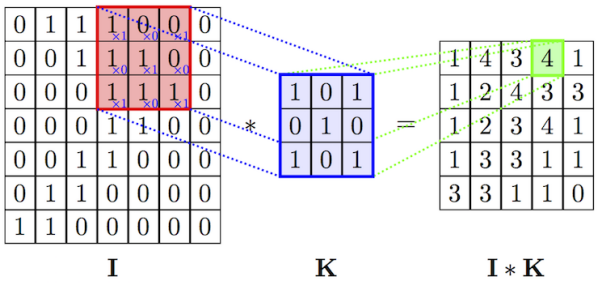
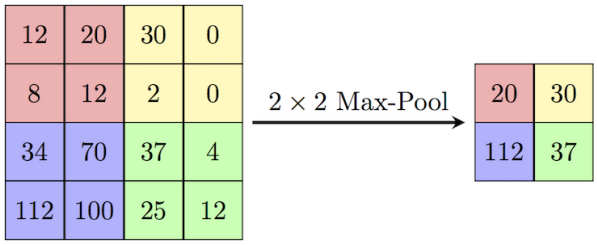


Figura 2.5: Estructura típica de una CNN. De (Peng et al., 2017).



(a) Convolución



(b) Max Pooling

Figura 2.6: Procesos de convolución (un filtro 3×3 con *zero-padding* a la derecha y *stride* 1) y *max-pooling* (2×2 con *stride* 2). De (Spark, 2017).

al número de características existentes (e.g. 10 para reconocimiento de números). La figura 2.6 muestra gráficamente los conceptos de capa convolucional y *max-pooling* con sus respectivos hiperparámetros.

Habitualmente, el flujo de información en las ANN fluye hacia delante de capas predecesoras hacia capas sucesoras. Sin embargo, existen arquitecturas en las que la información se retro alimenta, como en las redes neuronales recurrentes y recursivas (*Recurrent Neural Networks* y *Recursive Neural Networks*, RNN) o las redes neuronales residuales (*Residual Neural Network*, ResNet), una arquitectura más compleja donde la entrada a un conjunto de capas de la red es sumada a la salida que producen.

2.3. Entrenamiento de Redes Neuronales

Las redes neuronales pueden entrenarse para resolver una tarea específica de forma óptima, esto es, minimizar el error entre la salida esperada y la real. Existen distintas técnicas de aprendizaje automático (*Machine Learning*, ML), como el aprendizaje no supervisado (*Unsupervised Learning*) en el que se aprenden propiedades útiles de un *dataset*; el aprendizaje supervisado (*Supervised Learning*, SL), donde además cada ejemplo del conjunto se asocia con una etiqueta o salida conocida; o el aprendizaje por refuerzo (*Reinforcement Learning*, RL), un subconjunto del aprendizaje supervisado en el que no se interacciona con un *dataset* fijo, sino que hay una conexión entre el sistema de aprendizaje y el entorno. El aprendizaje profundo (*Deep Learning*, DL) es un subconjunto dentro del aprendizaje automático que hace referencia al aprendizaje de DNN, i.e. ANN con múltiples capas ocultas.

El objetivo último del aprendizaje es que el algoritmo funcione correctamente con nuevas entradas no vistas hasta el momento, y no únicamente con los *datasets* aprendidos. Mientras la optimización buscaría que el error entre la salida real de la red y la esperada se minimice durante el aprendizaje (*training error*, ε_t), el aprendizaje automático busca además que el error ante nuevos datos de entrada (*generalization error*, ε_g) sea mínimo. Dos de los grandes retos del aprendizaje automático son el sobreajuste (*overfitting*) y la sobregeneralización (*underfitting*). El primero de ellos ocurre cuando los pesos ajustan en exceso para los datos del entrenamiento y el algoritmo es incapaz de generalizar estos resultados a nuevos datos, provocando una gran diferencia entre los errores ε_t y ε_g. La sobregeneralización representa el caso opuesto, donde el modelo no es capaz de generar bajos errores de aprendizaje debido a, por ejemplo, insuficiencia de datos. El conjunto de técnicas para conseguir disminuir el error de generalización sin afectar al error de entrenamiento se conocen como regularización. El parámetro de capacidad del modelo indica si es más propenso a sobreajustar o sobregeneralizar. En el caso de una regresión polinomial, el grado del polinomio actuaría como capacidad, esto es, un polinomio de grado 7 sería más propenso a sobreajustar que uno de grado 2 para un conjunto de datos.

Las DNN son particularmente propensas al sobreajuste debido a su elevado nivel de abstracción. Tres de las técnicas de regularización más empleadas son las siguientes:

- Descomposición de las ponderaciones: consiste en añadir a la función de coste que se optimiza un término que penalice las ponderaciones elevadas. Este término puede ser la norma-2 ($\mathcal{L}^2$, *weight decay*) o la norma-1 ($\mathcal{L}^1$), que lleva a una solución en la que más ponderaciones resultan nulas (*sparsity*).
- Aumento (sintético) del *dataset*: aumentando la cantidad de datos, el propio algoritmo corrige el problema de sobreajuste.

- Marginalización (*Dropout*): durante el entrenamiento, se ignoran neuronas escogidas de forma arbitraria con una cierta probabilidad. Esto hace que las neuronas activas tengan que suplir la contribución de las neuronas marginalizadas. Como consecuencia, el grafo se vuelve menos sensible a las ponderaciones específicas de cada neurona y se consigue una mayor capacidad de generalización.

Uno de los algoritmos más difundidos para entrenar ANN mediante aprendizaje supervisado (i.e. cuando se conoce la señal de salida deseada) es *backpropagation* (BP). En primer lugar se propaga hacia delante la información en el grafo, de forma que se genere una señal de salida y pueda calcularse el error entre ésta y la salida esperada. A continuación, se propaga hacia atrás este término de error. Para ello debe calcularse la contribución de cada una de las ponderaciones al término de error (derivadas parciales). A cada neurona de cada capa se le asigna una parte del error total en función de su contribución, y se corrigen las ponderaciones para disminuir este error.

Considérese como ejemplo un entrenamiento en el que la función de coste está basada en mínimos cuadrados:

$$J(\theta,\ \mathcal{X}) = \frac{1}{2N} \sum_{i=1}^{N} \left[h_\theta(x(i)) - y(i)\right]^2 \tag{2.12}$$

O bien una función más compleja y más propensa a generalizar, que podría ser la conocida como entropía cruzada (Mitchell, 1997):

$$J(\theta,\ \mathcal{X}) = -\frac{1}{N} \sum_{i=1}^{N} y(i)\ \log\left(h_\theta(x(i))\right) + (1 - y(i))\ \log\left(1 - h_\theta(x(i))\right) \tag{2.13}$$

Donde $x(i) \in \mathcal{X}$ es el conjunto de entradas del *dataset* de entrenamiento, $y(i) \in \mathcal{Y}$ el conjunto de etiquetas, y $h_\theta(x)$ la salida de la red ante una cierta entrada x y con unos parámetros θ.

Para poder entrenar la red deben calcularse las derivadas parciales de la función de coste respecto a cada uno de los parámetros de la red. El algoritmo de BP permite realizar estos cálculos de una manera eficiente, minimizando la función de coste empleando el gradiente descendente (*Gradient Descent*, GD). Se puede dividir en varios pasos (Rojas, 1996, cap. 7):

1. *Feed-forward propagation.* Primero se asigna un valor aleatorio a cada parámetro y se propaga la información por la red, guardando el valor de la salida y de la función de activación de cada capa (tanto ocultas como de salida).
2. *Backpropagation.* En primer lugar se calcula el término de error en la capa final m, entre la salida esperada y la obtenida (δ_1^m). Se propaga hacia atrás este error hasta llegar a la primera capa (δ_j^k). Conocidos los términos δ, cuyo cálculo dependerá de la función de activación de

los nodos, se pueden evaluar las derivadas parciales de J respecto a las ponderaciones empleando la regla de la cadena como:

$$\frac{\partial J_d}{\partial w_{ij}^k} = \frac{\partial J}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k} \equiv \delta_j^k \; o_i^{k-1} \tag{2.14}$$

Donde w_{ij}^k es el peso del nodo j en la capa k para el nodo entrante i, a_j^k es la suma ponderada de las entradas (activación) al nodo j en la capa k, o_i^{k-1} la salida del nodo i en la capa $k-1$, y d el conjunto de datos del *dataset*. El gradiente total puede calcularse como la media de los gradientes para cada pareja de datos entrada-salida, esto es:

$$\frac{\partial J(\theta,\ \mathcal{X})}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial J_d}{\partial w_{ij}^k} \tag{2.15}$$

3. *Weight update*. Finalmente, las ponderaciones se pueden actualizar de acuerdo al gradiente total y al factor de aprendizaje, α:

$$\Delta w_{ij}^k = -\alpha \; \frac{\partial J(\theta,\ \mathcal{X})}{\partial w_{ij}^k} \tag{2.16}$$

Nótese que los *biases* se han introducido en los pesos para simplificar las expresiones, tal que el *bias* del nodo i-ésimo de la capa k-ésima se incorpora en los pesos $w_{0i}^k = b_i^k$.

Otra forma de llevar a cabo el problema de optimización anterior es emplear el gradiente descendente estocástico (*Stochastic Gradient Descent*, SGD).

2.3.1. Aprendizaje profundo

Las DNN permiten un nivel de abstracción muy elevado, lo cual también conlleva ciertas desventajas. Por ejemplo, cuando se emplea entrenamiento supervisado, aparecen mínimos locales en la función de coste y la optimización involucrada pasa a ser no convexa, pudiendo no converger el resultado a valores aceptables (especialmente al emplear GD). En la figura 2.7 se muestra un ejemplo de cómo la optimización convergería a resultados distintos dependiendo donde comenzase el proceso. Otro ejemplo es la conocida difusión de gradientes (*vanishing of gradients*). Al propagar los gradientes hacia atrás en BP, las primeras capas de la red cambian sus ponderaciones muy lentamente (i.e. aprenden muy poco). Esto resulta en capas finales muy entrenadas que prácticamente obvian a las primeras, generando los mismos resultados que si estas capas no existiesen y la señal de entrada estuviera ligeramente distorsionada. Si las ponderaciones se inicializan en valores muy bajos, las derivadas parciales tenderán a cero rápidamente, conduciendo al problema anterior. Si, por el contrario, las ponderaciones son muy elevadas, el entrenamiento se quedará estancado en mínimos locales.

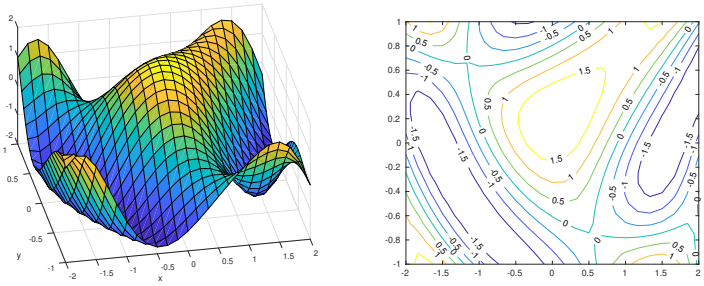


Figura 2.7: Representación de una función de coste no convexa donde el punto de inicio o semilla determinará la solución (mínimo local) y donde además existen puntos de silla (*saddle point*).

Un método que ha probado tener cierto éxito para entrenar DNN es el aprendizaje voraz por capas (*greedy layer-wise training*). Aunque este aprendizaje puede ser supervisado, habitualmente no lo es (e.g. *stacked autoencoders*). Este algoritmo consta de una fase donde se pre-entrena la red capa a capa, obteniendo una estimación inicial de las ponderaciones; y una segunda fase en la que se trata la red como un todo y se ajustan las ponderaciones (*fine-tunning*). El pre-entrenamiento permite que la optimización se inicialice en un punto cercano a la solución óptima, esto es, que la semilla se sitúe cerca del mínimo global en funciones no convexas. Una vez se ha pre-entrenado la DNN, el aprendizaje posterior es similar al de una ANN. Los problemas de atracción hacia mínimos locales y difusión de gradientes ya se han mitigado, por lo que la optimización es comúnmente realizada mediante GD o SGD (e.g. BP en el caso de aprendizaje supervisado).

Por lo tanto, el pre-entrenamiento permite evitar que la solución yazca en mínimos locales y también la difusión de los gradientes, al haber entrenado las primeras capas. Sin embargo, el pre-entrenamiento cayó en desuso (en la mayoría de las aplicaciones) al aparecer otros métodos que consiguen el mismo objetivo. Incluso redes muy profundas pueden entrenarse de forma satisfactoria empleando unidades de rectificación (ReLU) como funciones de activación, *batch normalization*, o *dropout*.

Otra forma de llevar a cabo el entrenamiento es mediante redes de creencia profunda (*Deep Belief Networks*, DBN). De nuevo, se trata de varias capas que se pre-entrenan una a una, pero en lugar de emplear *autoencoders* se emplean máquinas de Boltzmann restrictivas (*Restricted Boltzmann Machine*, RBM).

El aprendizaje por refuerzo (*Reinforcement Learning*, RL) se basa en el aprendizaje prueba-error, donde cada acción posible tiene asociada una recompensa y el agente trata de maximizar la recompensa que consigue; como se verá en el siguiente capítulo. El aprendizaje por refuerzo profundo (*Deep Reinforcement Learning*, DRL) aumenta el problema anterior a las DNN. En este caso, el algoritmo de BP puede ser usado pese a no ser un aprendizaje supervisado.

Capítulo 3

Deep Learning en SLAM Activo

El uso de redes neuronales profundas se ha extendido en muchas áreas de investigación en robótica, pero ¿es realmente posible aplicarlo a la toma de decisiones durante SLAM activo? Y en ese caso, ¿para qué sirve exactamente la red neuronal y qué aporta a las soluciones tradicionales?, ¿como debe entrenarse la red?, ¿es posible un aprendizaje generalista que permita trabajar en entornos cambiantes propios del problema?, ¿bastaría con conocer una serie de reglas de comportamiento? Estas son algunas de las preguntas que implícitamente se busca responder en este capítulo, en el que se formula el problema de toma de decisiones bajo el *framework* de los procesos de decisión de Markov, y se analizan las distintas soluciones, buscando siempre el posible potencial y la utilidad del aprendizaje profundo.

3.1. Toma de Decisiones

Como ya se ha visto, el problema de SLAM activo puede englobarse dentro de un marco de trabajo más general, los procesos de decisión de Markov parcialmente observables (POMDP); en los que se describe el proceso de toma de decisiones secuenciales cuando tanto las acciones como la adquisición de datos poseen incertidumbre. Recuérdese que estos procesos quedan formalmente definidos por la 7-tupla $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, \gamma)$, donde $\mathcal{S}$ es el conjunto de estados, $\mathcal{A}$ el de acciones, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto F_X[\mathcal{S}]$ el de probabilidades condicionales entre estados, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ la función de recompensa, Ω el conjunto de observaciones y $\mathcal{O} : \mathcal{S} \times \mathcal{A} \mapsto F_X[\Omega]$ sus probabilidades condicionales, y el factor de descuento $\gamma \in [0, 1]$. La función $F_X[\cdot] \in [0, 1]$ empleada anteriormente simboliza la distribución de probabilidad de $(\cdot)$. La probabilidad de cambiar de un estado s a otro s' contenidos en $\mathcal{S}$ vendrá dada por $\mathcal{T}(s, a, s') \doteq \mathbb{P}[s_{t+1} = s' | s_t = s, a_t = a]$; y la probabilidad de realizar una observación $o \in \Omega$ tras haber alcanzado un estado s será $\mathcal{O}(o, a, s) \doteq \mathbb{P}[o_t = o | s_t = s, a_t = a]$. La relación entre estos espacios puede apreciarse en la figura 3.1.

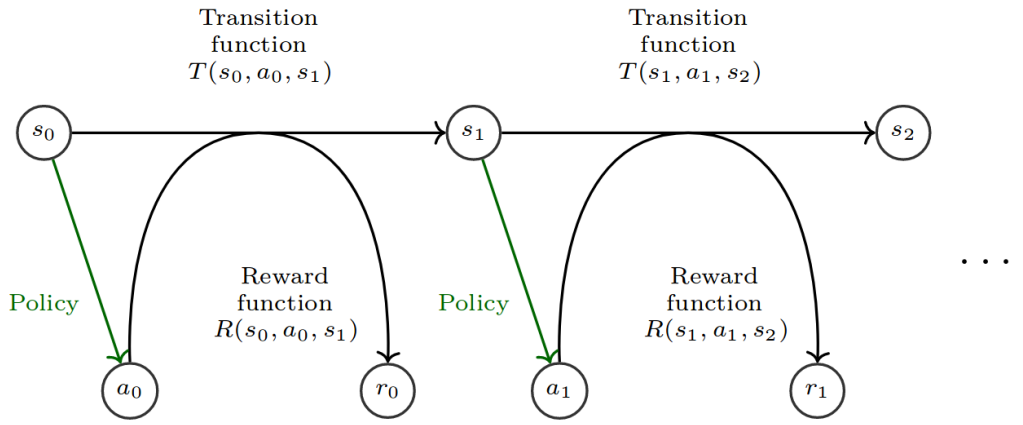


Figura 3.1: Ilustración de un MDP. De (François-Lavet et al., 2018).

3.1.1. Funciones de valor y reglas de comportamiento

En cada iteración del proceso, el agente ha de tomar una decisión basándose en los conjuntos de estados y acciones observables y observaciones pasadas. Su propósito es encontrar la regla que maximice una función objetivo definida sobre el histórico de conjuntos de acciones y estados (u observaciones, en caso de existir estados no observables), H. Estas funciones objetivo son las funciones de valor, $V(h) : H \mapsto \mathbb{R}$, y únicamente mapean el espacio H a un número real. Un conjunto $h \in H$ se preferirá sobre otro si su función de valor es más elevada. Una función de valor comúnmente usada que cuantifica el valor de la pareja estado-acción es la siguiente,

$$V(h) = R_t = \sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \tag{3.1}$$

Siendo t el instante temporal y $\mathcal{R}(s, a)$ la recompensa inmediata tras la transición de s a s' por haber realizado la acción a.

Una regla de comportamiento o *policy* determina el comportamiento del agente dado un conjunto de $\mathcal{A}$ y $\mathcal{O}$ (i.e. $\pi : H \mapsto \mathcal{A}$, o bien $\pi : \mathcal{S} \times \mathcal{T} \mapsto \mathcal{A}$). El objetivo del agente es buscar la regla óptima $\pi^* \in \Pi$, es decir, la regla cuyo valor esperado sea mayor. Este valor no es más que el valor esperado de los distintos conjuntos de acciones y observaciones inducidos por dicha regla:

$$V^{\pi}(h) \doteq \mathbb{E}_{\pi}[V|h] = \sum_{h \in H} V(h) \mathbb{P}\left[h|\pi, b^0\right] \tag{3.2}$$

Donde $\mathbb{E}[\cdot]$ es el valor esperado, $\mathbb{P}[\cdot]$ es la distribución de probabilidad y b^0 es la estimación *a priori* de los estados del sistema (función de la estimación anterior, la acción realizada y la observación actual). Y particularizando para $s \in \mathcal{S}$, donde $V^{\pi}(s) : \mathcal{S} \mapsto \mathbb{R}$:

$$V^{\pi}(s) = \mathbb{E}_{\pi,\mathcal{T}}\left[R_t|s_t = s\right] = \mathbb{E}_{\pi,\mathcal{T}}\left[\sum_{t=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\right] \forall s \in \mathcal{S} \tag{3.3}$$

La ecuación (3.3) puede reescribirse de forma que el valor esperado de una regla se pueda calcular mediante la siguiente recurrencia (ecuación de Bellman):

$$V_0^\pi(s) = \mathcal{R}\left(s, \pi(s,0)\right) \tag{3.4}$$

$$V_t^\pi(s) = \mathcal{R}\left(s, \pi(s,t)\right) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}\left[s, \pi(s,t), s'\right] V_{t-1}^\pi(s') \tag{3.5}$$

O, empleando el operador de valor esperado,

$$V_t^\pi(s) = \mathbb{E}_\pi\left[r_{t+1} + \mathbb{E}_\mathcal{T}\left[\gamma V_{t-1}^\pi(s')\right]\right] \tag{3.6}$$

Pudiendo simplificarse a la siguiente expresión para entornos deterministas:

$$V_t^\pi(s) = \mathcal{R}\left(s, \pi(s,t)\right) + \gamma V_{t-1}^\pi(s') \tag{3.7}$$

Sabiendo que la función de valor óptima será $V^*(s) = \max\limits_\pi V^\pi(s)$, el principio de optimalidad de Bellman permite calcular la función de valor óptima del instante t a partir de la del instante inmediatamente anterior,

$$V^*(s) = \max_{a \in \mathcal{A}} \left[\mathcal{R}\left(s,a\right) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}\left[s,a,s'\right] V^*(s')\right] \tag{3.8}$$

Esta última ecuación permite encontrar la función de valor óptima y, una vez conocida, también es posible calcular la *policy* óptima como $\pi^* = \arg\max\limits_\pi V^\pi(s)$.

La ecuación (3.5) permite conocer el valor esperado de una regla y es la base de los algoritmos *policy-iteration* que se explicarán más adelante. Los métodos *value-iteration* emplean la ecuación (3.8) para calcular las funciones de valor óptimas directamente.

La función Q (*Q-function* o *Q-value*), $Q : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$, representa el valor de ejecutar una cierta acción en un estado y seguir además una regla óptima. La relación entre esta y V viene dada por $V^*(s) = \max_a Q^*(s,a)$. De forma análoga a V:

$$Q(s,a) \qquad = \mathbb{E}_{\pi,\mathcal{T}}\left[R_t | s_t = s, a_t = a\right] \tag{3.9}$$

$$\therefore Q^*(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}\left[s,a,s'\right] V^*(s') \tag{3.10}$$

Esta función permite conocer directamente la *policy* óptima extrayendo la acción que genera la mayor recompensa para el estado s:

$$\pi^* = \arg\max_{a \in \mathcal{A}} Q^*(s,a) \ \forall s \in \mathcal{S} \tag{3.11}$$

Además, cabe destacar que una regla puede definirse de manera estocástica como $\psi = \mathcal{S} \mapsto \Pi(\mathcal{A})$. Frente a la definición anterior en la que un estado era mapeado a una acción, ahora cada estado es mapeado a la distribución de probabilidad de la acción (i.e. $\psi(s,a) \doteq \mathbb{P}[a_t = a | s_t = s]$ es la probabilidad de que a se ejecute en s). Esta definición representa únicamente una ventaja en el caso de POMDPs, en los que se puede convertir el espacio discreto $\mathcal{A}$ en un espacio continuo de distribuciones de probabilidad sobre el que es posible aplicar técnicas de optimización continuas (generalmente más sencillas que las discretas).

Nótese que puesto que en un POMDP no todos los estados son conocidos, será necesario redefinir el valor esperado de una regla (o de la *Q-function* para los algoritmos implícitos) en función de la estimación de dichos estados, b. Otra forma de resolución del POMDP es crear un MDP a partir de los estados estimados (*belief-state* MDP). Ver (Braziunas, 2003, cap. 4) y (Linh Thai, 2018).

3.1.2. Solución clásica al problema

La resolución del problema puede abordarse [1] calculando directamente la función de valor óptima (*value-iteration* o *value-based*) o calculando de forma iterativa la regla óptima con valor esperado mayor (*policy-iteration* o *policy-based*) y posteriormente calular la función de valor óptima. De esta forma, el problema puede resolverse mediante:

- *Value-iteration.* Este proceso comienza (i) inicializando arbitrariamente V (e.g. $V_0(s) = 0 \; \forall s \in \mathcal{S}$), y (ii) calculando de forma iterativa la función de valor $V(s)$ hasta que converge al valor óptimo V^*, ecuación (3.8). La regla se puede calcular entonces como se ha explicado en la sección anterior.
- *Policy-iteration.* Se debe (i) inicializar π y $V \; \forall s \in \mathcal{S}$, (ii) calcular el valor de la regla π_t (ec. 3.5) –*policy evaluation*–, y la *Q-function* para cada par estado-acción (ec. 3.10), y (iii) actualizar la regla según el valor óptimo de la función de valor (ec. 3.11) –*policy improvement*–.

A nivel computacional, los métodos *value iteration* suelen requerir un mayor número de iteraciones para converger. Sin embargo, en ellos únicamente hay que aplicar el operador de Bellman en cada iteración, mientras que en el caso de *policy iteration* hay que calcular el valor de la regla, o lo que es lo mismo, resolver un sistema lineal de ecuaciones, como se muestra a continuación. Partiendo de la ecuación (3.5), se puede expresar el cálculo del valor de una regla como:

$$v_\pi = r + \gamma P v_\pi \longrightarrow v_\pi = (\mathbb{I} - \gamma P_\pi)^{-1} \; r \tag{3.12}$$

[1] Nótese que existen además otras aproximaciones empleadas menos frecuentemente, como los métodos basados en la programación dinámica o en Monte Carlo.

Donde $v_\pi \in \mathbb{R}^{|\mathcal{S}|}$ es el vector de valores para cada estado, $r \in \mathbb{R}^{|\mathcal{S}|}$ el vector de recompensas y $P_\pi \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$ la matriz de probabilidades para cada transición bajo la *policy* π.

Otras soluciones no exactas al problema pueden encontrarse aplicando métodos de optimización numérica no lineal clásicos (métodos basados en el gradiente). La ventaja de estos radica en que las simplificaciones que se realizan (e.g. restricciones en las reglas), hacen que problemas de alta dimensionalidad (o incluso infinita) no resulten en *policies* infinitas ya que las reglas son optimizadas directamente en el espacio Π. Ver (Braziunas, 2003, cap. 5).

3.1.3. Aprendizaje por refuerzo

El aprendizaje por refuerzo (*Reinforcement Learning*, RL) es el área del aprendizaje automático que trata la toma de decisiones de agentes en un entorno, de forma que se maximice una cierta función de recompensa. En la figura 3.2 se muestra el proceso completo de RL. Se trata de un método válido para la resolución del proceso de decisión de Markov cuando éste no es conocido (los estados y las funciones de transición son desconocidas). Puede realizarse empleando distintos algoritmos y métodos, como los explicados en la sección anterior de optimización de la función de valor (basados en programación dinámica, en Monte Carlo o en *temporal differences*) o de la regla de comportamiento, ya sea determinista o estocástica; u otros métodos más complejos como los métodos *advantage actor-critic* en los que de alguna forma se combinan los dos anteriores (*value-based critic* que evalúa cómo de buena es la acción elegida y *policy-based actor* que controla el comportamiento del agente) y se emplea la función "ventaja" para estabilizar el aprendizaje empleando gradientes. Dicha función indica la mejora que supondría tomar una cierta acción a desde un estado s respecto del valor promedio de dicho estado, $A(s,a) = Q(s,a) - V(s)$.

Tanto los métodos *policy-* como *value-based* pertenecen a una categoría superior denominada *model-free* RL, donde el agente se centra en desarrollar una función que mapee el estado en la mejor acción posible y no en modelar el MDP (i.e. el entorno), suponiéndose desconocido, y aprendiendo directamente mediante la interacción con él. Frente a esta aproximación, se encuentran los métodos *model-based* en los que en primer lugar se aprende el modelo del entorno (modelo probabilístico) y posteriormente se toman decisiones en base a éste. Nótese por tanto que en estos métodos el agente es capaz de hacer predicciones de estados y recompensas futuras antes de realizar una acción, esto es, simular transiciones. Esto resulta en una eficiencia mayor pero también en la posibilidad de aparición de errores derivados de un modelo incorrecto. Ver figura 3.3.

En el anexo A se explican dos algoritmos habituales en RL: *Q-learning* y *Policy Gradient*.

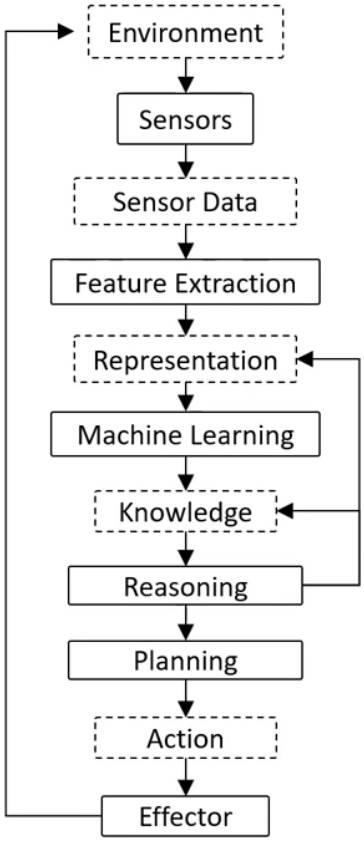


Figura 3.2: Proceso de *(Deep) Reinforcement Learning*. La información del entorno puede introducirse como *input* al problema mediante el uso de distintos sensores (e.g. cámaras, sensores inerciales), que, con distintas técnicas (e.g. redes neuronales) codifican esta información de forma que el agente sea capaz de entenderla y reconocer ciertas características o patrones. Finalmente, a partir de este razonamiento, el agente debe ser capaz de tomar una decisión que generará un cambio en el estado del agente y en su entorno. Adaptado de (Fridman, 2019).

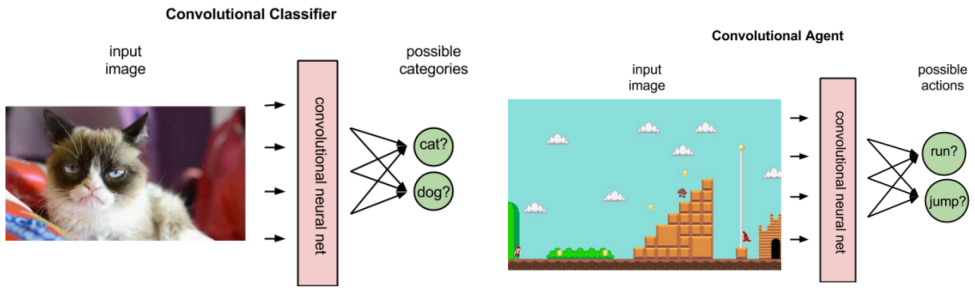
Figura 3.3: División de métodos de *(Deep) Reinforcement Learning* y su eficiencia en cuanto al número de datos necesarios para aprender. Adaptado de (Fridman, 2019).

3.2. Aprendizaje por Refuerzo Profundo

3.2.1. Introducción

El aprendizaje por refuerzo profundo (*Deep Reinforcement Learning*, DRL) hace uso de redes neuronales profundas (DNN) para resolver el problema de RL. Como todo sistema de ANN, emplea coeficientes para aproximar una función que relacione las entradas del sistema con sus salidas, y su aprendizaje consiste en el ajuste correcto de estos coeficientes de forma iterativa basándose en gradientes y en el algoritmo de *backpropagation*. En el ámbito de RL, DNN o CNN pueden emplearse para reconocer estados del sistema (o incluso estimar aquellos que están ocultos) y escoger la acción más ventajosa. El aprendizaje por refuerzo puede considerarse similar al aprendizaje supervisado, pero en lugar de asignar etiquetas durante el proceso de aprendizaje, se asignan posibles acciones, i.e. $a = \pi(s)$; tal y como se puede ver en la figura 3.4. Al fin y al cabo, todos los tipos de aprendizaje podrían considerarse "supervisados" por una función de pérdida u objetivo. En DRL, las redes neuronales son el agente capaz de mapear los estados o parejas estado-acción a recompensas, o de forma equivalente, a funciones ya vistas como V o Q. Entre las ventajas de emplear DNN se encuentra la posibilidad de resolver problemas que con métodos clásicos (e.g. tabulares) resultan irresolubles. Tómese como ejemplo el algoritmo *value-based Q-learning*. Se puede apreciar rápidamente que la tabla generada que relaciona cada pareja estado-acción con su *Q-value* sería de un tamaño excesivo para, por ejemplo, problemas en los que el cambio de un píxel de una imagen generase un nuevo estado. Las DNN permiten solucionar este problema simplificando la solución de los métodos tabulares: en lugar de disponer de la *Q-function* exacta, se dispone de una aproximación de ésta.

Algunas características que quizá merezca la pena enfatizar sobre (D)RL son que (i) la regla de comportamiento óptima se encuentra mediante prueba-error, siendo la recompensa el único dato del que el agente dispone, (ii) cada observación depende de las acciones realizadas por el agente, y pueden estar



(a) Aprendizaje supervisado (b) Aprendizaje por refuerzo

Figura 3.4: Ejemplos del distinto uso de CNN en aprendizaje supervisado y por refuerzo. De (SkyMind, 2018).

fuertemente correladas, y (iii) las consecuencias de una acción pueden prolongarse en el tiempo y no materializarse inmediatamente (*long-term temporal dependency*) (Arulkumaran et al., 2017).

En los algoritmos de DRL, en lugar de modelar los conjuntos de probabilidades $\mathcal{O}(s_t, a_t) \mapsto o_t$ y $\mathcal{T}(s_t, a_t) \mapsto s_{t+1}$ se modela una función de transición combinada $\mathcal{T}_\mathcal{O}(s_t, a_t, o_t, r_t, \theta_\mathcal{T}) \mapsto s_{t+1}$. En los algoritmos *policy-based* también se modela una función de valor $V(s_t, a_t, o_t, r_t, \theta_V) \mapsto V_t(\theta_V)$ y una función de regla $\pi(a_{t+1}|s_t, a_t, o_t, r_t, \theta_\pi) \mapsto \pi_t(a_{t+1}|\theta_\pi)$. El objetivo del DRL es estimar los parámetros $\theta = \theta_\mathcal{T} \cup \theta_\pi \cup \theta_V$ que maximicen la recompensa futura descontada esperada, o lo que es lo mismo, $\theta^* = \arg\max_\theta \mathbb{E}[R_t]$. Las tres funciones modeladas comparten algunos de sus parámetros tal que $\theta_\mathcal{T} \subset \theta_\pi \cap \theta_V$ (Banerjee et al., 2018).

En el anexo B se explica el algoritmo *Deep Q-Networks* (DQN), extensión de *Q-learning* con DNN.

3.2.2. DRL en exploración

El uso de DNN para representar las reglas o estrategias en RL es relativamente reciente. Permite una representación muy detallada, pero tiene grandes inestabilidades durante el aprendizaje online. Distintas soluciones se han propuesto para estabilizar el algoritmo, como las basadas en *Experience Replay* (ER), que guardan en una memoria de un cierto tamaño (Liu & Zou, 2018) experiencias pasadas y en cada iteración muestrean un subconjunto de ellas para actualizar los parámetros del agente. Cada experiencia podría ser, por ejemplo, la 4-tupla (s_t, a_t, r_t, s_{t+1}). De esta forma se evita la correlación entre datos sucesivos, evitando mínimos locales y el sobreajuste. Este tamaño del muestreo ha probado ser clave en el aprendizaje, teniendo su incremento el mismo efecto que la disminución del *learning rate*, α (Smith et al., 2017).

(Volodymyr Mnih et al., 2015) es uno de los primeros trabajos donde se emplea ER de forma satisfactoria, superando a todos los algoritmos del momento. Los mismos autores presentan una solución alternativa en (Volodymyr Mnih et al., 2016). El algoritmo A3C (*Asynchronous Advantage Actor Critic*) propone la ejecución asíncrona de múltiples agentes en paralelo, y ha sido empleado en numerosos trabajos ya que consigue la estabilización del aprendizaje sin el uso de ER, como en (Lei et al., 2017; J. Zhang et al., 2017; Zhelo et al., 2018) o (Mirowski et al., 2016), donde se aumenta además el aprendizaje con objetivos auxiliares (e.g. la estimación de profundidad o la detección de lazos) para mejorar su funcionamiento.

Los métodos basados en este algoritmo representan una nueva aproximación al problema con respecto a las vistas hasta ahora, ya que combina las funciones de valor (*critic*) con una representación implícita de las reglas (*actor*). De esta forma, la función de valor se emplea como *feedback* para la optimización de la regla: tras ejecutar una acción, se genera una recompensa y un

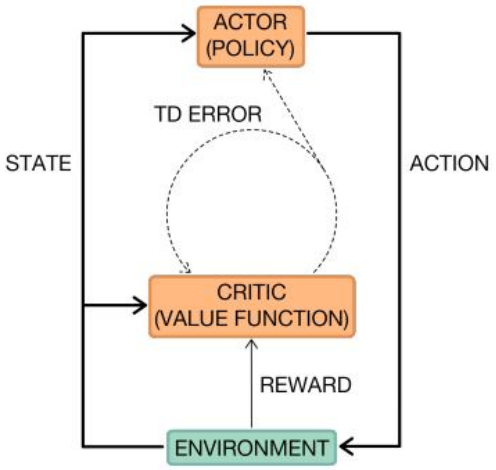


Figura 3.5: Arquitectura *actor-critic*. De (Arulkumaran et al., 2017).

nuevo estado en función del entorno, que permiten calcular la función de valor. Esta función de valor junto con el estado son empleados para calcular la regla óptima que generará una nueva acción. Tanto la función de valor como la regla se realimentan con el error TD (*temporal difference*), $\delta_{t-1} = r_t + \gamma V_t - V_{t-1}$. Este término suele referirse como ventaja o *advantage* y puede representarse como $A = Q(s, a) - V(s)$, o de forma equivalente, como $A \approx R - V(s)$ si los *Q-values* no son calculados en el algoritmo (ver figura 3.5). Nótese que tanto la función de valor como la regla son calculadas en una red neuronal.

En (Mirowski et al., 2016) se consigue entrenar la red para encontrar objetivos aleatorios en un mapa con distintas posiciones iniciales del agente, pero no se presentan resultados en distintos mapas, por lo que se plantea la pregunta de si se podría generalizar el aprendizaje a mapas no conocidos. Oh et al., (2016), en contraste, sí generalizan los resultados en mapas de diferentes dimensiones obteniendo buenos resultados, pero en este caso el agente no busca un *goal* concreto sino que es entrenado para escoger entre varios basándose en el historial de observaciones. En Banerjee et al., (2018) se emplea una aproximación similar a la de Mirowski, pero con una gama de experimentos más amplia, intentando responder a la pregunta: ¿realmente aprenden los algoritmos de DRL a explorar?. Al generalizar el aprendizaje (llevado a cabo en $N = \{10, 100, 500, 1000\}$ mapas) a mapas desconocidos se observa que se está explotando y no explorando [2] el entorno, siguiendo, por ejemplo, estrategias de seguimiento de paredes. En experimentos simples se detecta que únicamente se explora aleatoriamente el mapa, en lugar de buscar el camino más corto hacia el objetivo. Los resultados en distintos mapas muestran que el camino más corto únicamente se escoge en aproximadamente el 50 % de ocasio-

[2]Una decisión puede tomarse de forma que se escoja la mejor acción posible con la información actual disponible (explotación) o de forma que se mejore la información (exploración). De esta forma, un sacrificio a corto plazo puede llevar a mejoras de comportamiento a largo plazo. Este dilema se conoce como *exploration-exploitation* y debe llegarse a un compromiso entre ambas decisiones para optimizar la exploración de un mapa.

nes (i.e. no presenta una significativa mejora sobre un movimiento aleatorio). Este trabajo es uno de los pocos en los que que se evalúa el aprendizaje en mapas desconocidos, probando que realmente el agente no está aprendiendo a explorar. Los mismos autores presentan en (Dhiman et al., 2018) resultados de generalización de DRL, concluyendo que los algoritmos (A3C) funcionan bien únicamente cuando el entorno es similar al de entrenamiento.

En (Zhelo et al., 2018) se realizan experimentos de generalización en nuevos mapas comparando diferentes algoritmos:

(a) A3C: dos capas convolucionales y dos totalmente conectadas (FCL, *Fully Conneced Layer*) seguidas de sus respectivas no linealidades (ELU, *Exponential Linear Unit*) para codificar las lecturas de los sensores, seguidas de una capa con 16 LSTM (*long short term memory*) y finalmente una capa lineal seguida de *softmax* (para producir las probabilidades de cada acción) y otra capa lineal (para generar la función de valor)

(b) A3C incluyendo el término de pérdida de entropía [3] en el cálculo de los gradientes ($d\theta_\pi$). El algoritmo A3C actualiza los parámetros de las *policy* y *value networks* (θ_π y θ_v) de forma que se maximice la recompensa esperada. En este trabajo se ha aumentado la actualización de la *policy* de forma que la entropía de esta sea tenida en cuenta y se evite la convergencia hacia *policies* determinísticas subóptimas, motivando al agente a tomar acciones "más impredecibles":

$$d\theta_\pi = \underbrace{\nabla_{\theta_\pi} \left(\log \pi(a_t|s_t, \theta_\pi)\right) \left(R_t - V(s_t, \theta_v)\right)}_{\text{término típico A3C}} + \underbrace{\beta \nabla_{\theta_\pi} \left(H\left(\pi(a_t|s_t, \theta_\pi)\right)\right)}_{\text{término entropía}} \tag{3.13}$$

(c) A3C y además un ICM (*Intrinsic Curiosity Module*) que contiene tres capas totalmente conectadas seguidas de no linealidades para codificar las lecturas de los sensores, una capa lineal con *softmax* (para calcular la acción estimada a partir de esta codificación) y dos capas lineales seguidas de ELU más otra capa lineal que a partir de la acción y las observaciones codificadas predice la observación siguiente

(d) A3C con entropía e ICM

En mapas conocidos, (d) es capaz de superar todos los mapas con éxito (i.e. encontrar el *goal* en N pasos), mientras que los demás algoritmos fallan en algunas ocasiones. La inclusión de memoria (capas LSTM) incrementa su éxito es más de un 25 % y hace converger a una buena regla de comportamiento. En mapas desconocidos, sin embargo, incluso este algoritmo falla en numerosas ocasiones, en función de la complejidad del mapa y, probablemente, de su parecido con los empleados durante el entrenamiento. No se reportan resultados de *random-walks* en los mismos mapas, pero cabría esperar peores resultados. Los mínimos locales (caminos no mínimos hasta el objetivo) parecen evitarse

[3] La entropía de una *policy* puede entenderse como la aleatoriedad de las acciones que el agente toma. Cuanto mayor es la entropía, más arbitrarias son las acciones escogidas.

al emplear el módulo de curiosidad ya que la recompensa intrínseca atrae al agente hacia estados nuevos y difíciles de predecir y por tanto disponen de mayor información de la que tendría un agente aleatorio (que podría estar revisitando las mismas áreas todo el rato). El módulo de curiosidad contribuye en la recompensa añadiendo el siguiente término a una recompensa extrínseca típica,

$$R^i = \frac{1}{2}||\hat{\phi}_{t+1} - \phi_{t+1}||_2^2 \tag{3.14}$$

Donde ϕ_{t+1} es el estado s_{t+1} codificado en sus correspondientes *features*, y $\hat{\phi}_{t+1}$ la predicción de éstas realizada por el módulo de curiosidad; de forma que se recompensan más aquellos estados difíciles de predecir y no conocidos.

La inclusión en la función de recompensa de diferentes componentes afecta a la capacidad de explorar del agente significativamente. Incluir la incertidumbre del robot o una métrica de ésta (i.e. criterios de optimalidad) favorecería la navegación óptima, ayudando a escoger al agente las acciones con menor incertidumbre asociada. Sin embargo, ¿es posible introducir esta recompensa en algoritmos de DRL?. Sean un sistema de SLAM tradicional capaz de devolver el mapa y la posición del agente en éste a partir de e.g. imágenes; y una red neuronal profunda capaz de resolver el problema de toma de decisiones, e.g. DQN, tomando como entrada el estado del robot, y devolviendo como salida una dirección de movimiento (e.g. el espacio finito $\mathcal{A} = \{\leftarrow,\ \rightarrow,\ \uparrow,\ \downarrow\}$). Parece obvio pensar entonces, que, efectivamente, el agente podría ser recompensado con una métrica de la incertidumbre (Σ). La función recompensa tradicional simplemente debería incluir un término asociado a esta incertidumbre que podría ser, empleando el criterio *D-opt*, $R^u \propto^{-1} \det(\Sigma)$.

¿Sería posible introducir esta métrica de Σ de otra manera en la red? ¿Qué pasaría si se tuviera en cuenta en la propia red modificando la función objetivo, y por tanto la actualización de sus pesos θ? Ya se ha visto que algunos algoritmos como A3C incluyen la entropía de la *policy* en la actualización de los pesos θ_π, i.e. $H(\pi) = -\sum(\pi(s)_k \log \pi(s)_k)$ siendo $\pi(s)_k$ la probabilidad de la acción k-ésima en el estado s. Las funciones objetivo del algoritmo serán entonces,

$$\mathcal{L}_v = \sum(R - V(s))^2 \tag{3.15}$$

$$\mathcal{L}_\pi = -\log(\pi(a|s))A(s) - \underbrace{\beta H(\pi)}_{\text{tér. regularización}} \tag{3.16}$$

Y la función global:

$$\mathcal{L} = \frac{1}{2}\mathcal{L}_v + \mathcal{L}_\pi = \frac{1}{2}\sum(R - V(s))^2 - \log(\pi(a|s))A(s) - \beta H(\pi) \tag{3.17}$$

Entonces, puede verse que implícitamente se está introduciendo la recompensa en el problema de optimización, por lo que sería equivalente.

En este caso se estaría motivando la visita de estados novedosos y se optimizaría la exploración conforme se entrenase el agente, incluso revisitando áreas conocidas para cerrar lazos (i.e. reducir la incertidumbre).

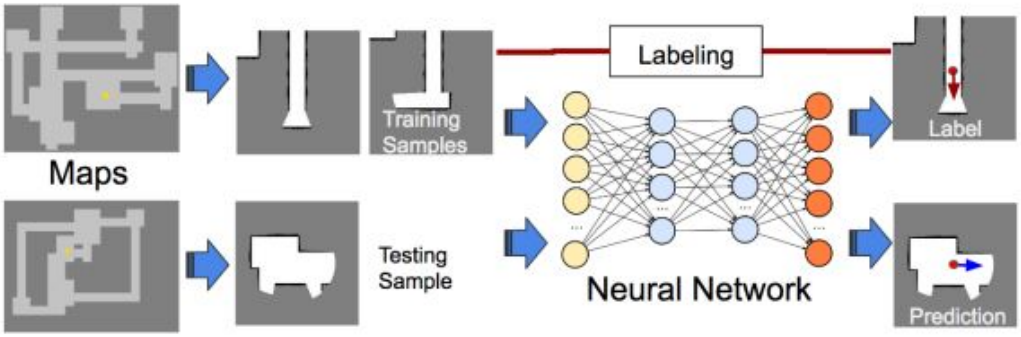


Figura 3.6: Fase de aprendizaje offline y funcionamiento del marco empleado en (Bai et al., 2017).

Las aproximaciones en las que se guía la exploración mediante la incertidumbre (*uncertainty-driven exploration*) son reducidas y habitualmente empleadas en *model-based* DRL para manipulación, de forma que esta incertidumbre puede introducirse en el modelo. Algunos ejemplos son (Depeweg et al., 2017) o (Büchler et al., 2018) donde se emplea la incertidumbre para promover o evitar, respectivamente, la exploración de lugares con alta incertidumbre, o (Chua et al., 2018) en el que se propone una estrategia de DL no supervisado.

En (Bai et al., 2017) se propone una solución basada en aprendizaje supervisado profundo en lugar de RL, como se había hecho en todos los trabajos previamente mencionados. Se etiquetan mapas 2D de forma que una red neuronal sea capaz de predecir la acción más valiosa a partir de la entropía de Shannon (ver figura 3.6). Se emplean y comparan distintas redes neuronales existentes como AlexNet (Krizhevsky et al., 2012), GoogleNet (Szegedy et al., 2015) y ResNet (He et al., 2016), entre otras. Pese a ser un ejemplo de entrenamiento supervisado, este trabajo emplea directamente una función de utilidad como herramienta para aprender a tomar decisiones.

La *Q-function* puede emplearse en lugar de V para conocer el valor de cada acción, lo que se denomina *Q-learning*, como ya se ha visto. Esto permite resolver problemas en los que la función de transición no es conocida (*model free*); aunque presenta dificultades con espacios de estados de gran dimensión, puesto que la dimensión de $Q(s, a)$ es incluso mayor que la de $V(s)$ pudiendo existir espacios $\mathcal{A}$ continuos. Las *Deep Q-Networks* (DQN) emplean redes neuronales para el cálculo (aproximado) de *Q-functions*, esto es: $Q(s, a, \theta) \approx Q^{\pi}(s, a)$ (ver figura 3.7 y anexo B).

Este tipo de agentes, introducidos por Google Deep Mind en (Volodymyr Mnih et al., 2015), son un buen ejemplo de resolución de MDP con $\mathcal{S}$ de elevada dimensión empleando DRL y SGD (*Stochastic Gradient Descent*), de forma que futuras recompensas sean maximizadas. Sin embargo, ésta, puede considerarse una aplicación limitada, ya que imágenes 2D *greyscale* son la única entrada a la red y $\dim \mathcal{A} = 3$.

Lei y Ming (2016) consiguen explorar un entorno desconocido sencillo

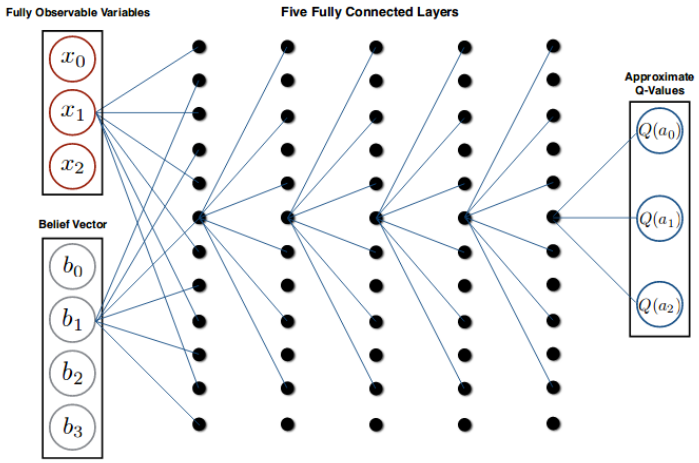


Figura 3.7: Ejemplo de la estructura de una DNN capaz de estimar *Q-functions* a partir de los estados conocidos y los estimados. De (Egorov, 2015).

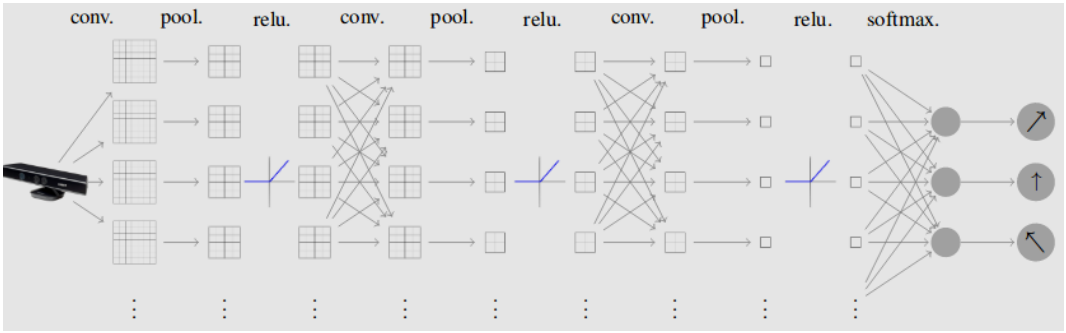


Figura 3.8: Estructura de la DQN propuesta en (Lei & Ming, 2016).

en Gazebo evitando obstáculos, empleando una DQN (ver figura 3.8). Una CNN entrenada offline de forma supervisada se usa para extraer los mapas de características (*feature maps* o *activation maps*) de imágenes RGB-D, que se alimentan a otra red totalmente conectada que se entrena online para calcular las *Q-functions*, y escoger la mejor acción. A pesar lograr la convergencia de la red, únicamente se resuelve la exploración en entornos ya navegados durante el entrenamiento, generando, al menos, preguntas en cuanto a la capacidad de generalización. La recompensa se modela mediante una función sencilla diseñada para evitar obstáculos, de forma que únicamente penaliza el acercarse a éstos.

Karkus et al., (2017) proponen otra solución al problema de navegación autónoma en el que se sustituye la CNN por una RNN con capas LSTM, de forma que permita abordar la observabilidad parcial, hecho olvidado en la mayoría de trabajos. En la red se codifica tanto el entorno como el algoritmo de aprendizaje. Consta de (i) una etapa de filtrado en la que se obtiene la estimación del instante $(t+1)$ a partir de la estimación, acción y observación

del instante t, i.e. un filtro Bayesiano,

$$b_{t+1}(s) = \eta \, \mathcal{O}(s, o) \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') b_t(s') \tag{3.18}$$

(siendo η una constante de normalización) y (ii) una etapa de planificación que ejecuta un método *value-iteration* mediante el que encuentra la acción preferida para $(t + 1)$. En primer lugar se calculan los *Q-values*, y posteriormente se actualizan los valores de V, según

$$Q_{k+1}(s, a) = \mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}\left[s, a, s'\right] V_k(s') \tag{3.19}$$

$$V_k(s) = \max_a \; Q_k(s, a) \tag{3.20}$$

La ecuación (3.19) se codifica mediante una serie de convoluciones con $\mathcal{A}$ filtros, seguidos de una operación de suma con tensor recompensa; mientras que la eq. (3.20) se consigue con capas *max-pooling*. Estas dos ecuaciones – denominadas como actualización de Bellman – se repiten de forma iterativa apilando las capas k veces. Tras k iteraciones se conocen los *Q-values* aproximados para cada par acción-estado, que, ponderados con la estimación (*belief*), permiten conocer el valor de cada acción y escoger así la más ponderada.

$$q(a) = \sum_{s \in \mathcal{S}} Q_k(s, a) b(s) \tag{3.21}$$

Las reglas aprendidas generalizan satisfactoriamente a nuevos entornos de mayor dimensión o con una configuración distinta, pero como elementos o características similares. Esto es gracias a que la red no aprende una regla óptima para los casos de entrenamiento sino un modelo de planificación más complejo. En algunos casos, QMDP-net llega a superar al algoritmo QMDP que está codificando, pese a que éste conoce el modelo real del POMDP, debido a que la red redefine los valores de las recompensas de tomar ciertas decisiones permitiendo volver a lugares ya visitados, por ejemplo. Hay que remarcar que durante el proceso se itera sobre la totalidad del espacio de estados (*value-iteration*), hecho factible únicamente por la baja dimensionalidad de este en los experimentos realizados.

La inclusión de memoria en estas redes permite disponer de una representación interna del entorno, mejorando la exploración. Sin embargo, en escenarios de larga duración, con la creciente demanda de memoria, la capacidad expresiva de la red se reduce potencialmente (Graves et al., 2016). Una memoria externa evitaría mezclar en los pesos de la red la memoria y los distintos algoritmos de cálculo. Existen tres arquitecturas de memoria externa para redes profundas: *Neural Turning Machine* (NTM), *Differentiable Neural Computer* (DNC) y las redes de memoria (*memory networks*), aunque estas últimas no aprenden qué deben escribir en la memoria (Graves et al., 2014; Graves et al., 2016; Oh et al., 2016).

J. Zhang et al., (2017) embeben en una memoria externa una representación interna del entorno, a partir de la que una red neuronal es capaz de ejecutar tareas de planificación y SLAM, incluso *long-term*. En cada *step* primero se actualiza la memoria $\mathcal{N}^t$ y posteriormente se realiza la toma de decisiones, que tras ser procesado por distintas capas de la red resulta en π^t y V^t. Esta aproximación, denominada Neural-SLAM, es validada en entornos 2 y 3D en Gazebo (donde la física subyacente y los modelos de ruido de sensores son mucho más realistas que en los entornos de simulación empleados en otros trabajos) con cuatro posibles acciones. Se consiguen resultados superiores a los conseguidos con *random-walks* y otras aproximaciones del algoritmo A3C, como (Mirowski et al., 2016). Se incluyen además pruebas de generalización en escenarios de mayor dimensión que los del entrenamiento. En la tabla 3.1 se han agrupado los resultados obtenidos. En primer lugar, se observa una mejora entre un movimiento aleatorio y el algoritmo propuesto en (Mirowski et al., 2016). Sin embargo, como ya explicaba (Banerjee et al., 2018), la mejora es sutil. Se consigue finalizar únicamente un 10 % de los escenarios en menos de 750 iteraciones y no se presenta la información del resto de escenarios para su comparativa. Las dos capas LSTM de la RNN podrían estar almacenando datos de la odometría del robot, incitándole a moverse en un vecindario cercano sin volver a lugares lejanos ya visitados o a realizar estrategias de seguimiento de paredes (como ya se ha comentado). La mejora del uso de una memoria externa, por otra parte, sí representa una mejora sustancial al disponer de una representación del entorno. Los experimentos en Gazebo consiguen ser satisfactorios pero no se presentan resultados de generalización en entornos desconocidos, trabajo que se plantea a futuro junto con la inclusión de recompensas intrínsecas de forma similar a (Zhelo et al., 2018).

Agente	**Steps**	**Recompensa**	**Tasa éxito**
Aleatorio	5531 ± 4299	-596 ± 505	-
A3C (Mirowski et al., 2016)	683 ± 201	-15 ± 11	0.10
Neural-SLAM (J. Zhang et al., 2017)	175 ± 175	13 ± 9	0.92

Tabla 3.1: Resultados aproximados de generalización en los experimentos 2D de (J. Zhang et al., 2017), donde la tasa de éxito se refiere a la capacidad de explorar un entorno en menos de 750 *steps*.

Otras soluciones de vanguardia de *Deep Q-learning* alternativas a DQN son DBQN (*Deep Belief Q-network*), DRQN (*Deep Recurrent Q-network*), DDRQN (*Deep Distributed Recurrent Q-network*) o ADRQN (*Action-based Deep Recurrent Q-network*), propuestas respectivamente en (Egorov, 2015; Hausknecht & Stone, s.f.; Foerster et al., 2016; Zhu et al., 2018). En la tabla 3.2 se presenta una breve comparativa de estas aproximaciones. Nótese que en ADRQN se introducen parejas acción-observación acopladas a la red, mejorando significativamente la estimación con respecto a DDRQN. La figura 3.9 muestra de forma muy clara la estructura de una ADRQN, con sus distintas capas, entradas y operaciones; pudiendo verse la metodología de cálculo,

Aproximación	Entrada	Problema abordado
DQN	s_t	*model-free* MDP
DBQN	b_t	*model-based* POMDP
DRQN	$(o_1, ..., o_t)$	*model-free* POMDP
DDRQN	$(a_0, ..., a_{t-1})(o_1, ...o_t)$	*model-free* POMDP
ADRQN	$(a_0, o_1), ..., (a_{t-1}, o_t)$	*model-free* POMDP

Tabla 3.2: Comparativa entre distintas aproximaciones de *Deep Q-learning*.

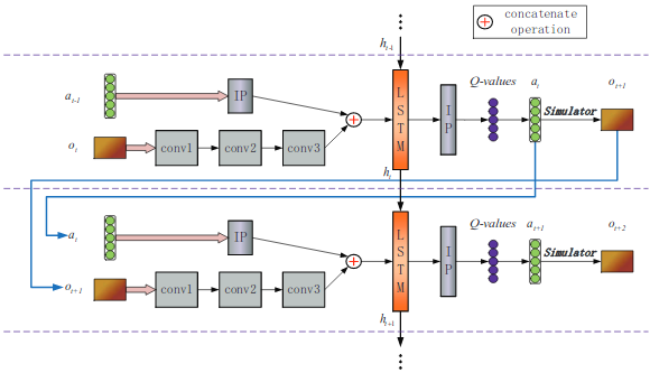


Figura 3.9: Estructura de la ADRQN propuesta en (Zhu et al., 2018), donde IP (*inner product*) es equivalente a FCL (*fully connected layer*).

que, a rasgos generales, es similar en todas las aproximaciones presentadas.

En (F. Chen et al., 2019), se emplea una aproximación de DRL con recompensas basadas en la entropía del mapa (*mutual information*) para explorar de forma eficiente entornos 2 y 3D. Este trabajo, publicado durante la realización de esta memoria, es similar a las implementaciones que se verán en el capítulo 4, donde la función recompensa es aumentada con una métrica de la incertidumbre para favorecer la exploración. Es el único trabajo que se ha encontrado en el que se emplea una DQN con recompensas basadas en la covarianza, y presenta resultados en los que se supera a algoritmos tradicionales de SLAM activo, pero siempre en entornos de tipo *grid-world*.

Capítulo 4

Implementación y Evaluación de RL en Exploración

Los contenidos de este capítulo están organizados de la siguiente manera:

- En primer lugar se describen las herramientas empleadas para la implementación de algoritmos y el hardware disponible (sección 4.1).
- En la sección 4.2 se presentan los escenarios de simulación empleados y sus características.
- Finalmente se muestran los detalles de las distintas implementaciones y los resultados obtenidos (secciones 4.3 en adelante).

4.1. Herramientas

A continuación se describen brevemente algunas de las herramientas empleadas en este trabajo. Los experimentos se han llevado a cabo en dos terminales distintos para agilizar el proceso, ambos con Ubuntu 16.04, ROS Kinetic y Gazebo 7.15. El primero de ellos (T1 de ahora en adelante) dispone de 16GB de RAM DDR3, CPU de 8 núcleos (i7-4771 @ 3.5GHz) y GPU Nvidia GTX-660 2GB; y el segundo (T2) de 32GB de RAM DDR4, CPU de 4 núcleos (i7-7500U @ 2.7GHz) y GPU Nvidia Quadro M520 2GB.

4.1.1. Python, TensorFlow y Keras

TensorFlow[1] es una librería de código abierto para ML desarrollada originalmente en 2015 por Google Brain que funciona con CPUs (*Central Processsing Units*), GPUs (*Graphics Processsing Units*) e incluso TPUs (*Tensor Processsing Units*). Esta librería empleada para cálculo numérico está basada en grafos

[1] https://www.tensorflow.org/

donde los nodos representan las operaciones y los arcos los tensores que fluyen entre ellos. Pese a que permite trabajar en Python las operaciones son realmente realizadas en código de alto rendimiento en C++.

Keras[2] es una API (*Application Programming Interface)* de alto nivel de redes neuronales que es capaz de ejecutarse sobre otras librerías de optimización o cálculo matemático como Tensorflow, CNTK[3] o Theano[4]. Se trata de una librería Python para aprendizaje profundo que permite crear modelos de los agentes de una forma sencilla con TensorFlow en *backend*. Esto es, una libería que no se encarga de operaciones de bajo nivel como productos de tensores o convoluciones, sino de facilitar la creación de modelos de DL mediante bloques.

En estre trabajo se han empleado tanto Python 2.7.12 como Python 3.5.2, TensorFlow 1.13.1 (CPU y GPU), Theano 1.0.4 y Keras 2.1.2.

4.1.2. OpenAI Gym

OpenAI Gym[5] es un paquete para el desarrollo y la comparación de distintos algoritmos de RL, desarrollado por OpenAI. Contiene un conjunto de entornos que van desde problemas de control clásicos como el péndulo invertido hasta otros más complejos como los juegos de Atari. Además, proporciona herramientas para la creación de espacios de acciones, de estados, o para la recolección de observaciones, en Python.

4.1.3. Gym Gazebo

Gym-Gazebo[6] es un software complejo que extiende las librerías de OpenAI Gym para su uso conjunto con ROS (*Robot Operating System*)[7] y Gazebo[8]. Contiene entornos de Gazebo complejos en los que simular comportamientos cercanos al mundo real. Este proyecto de Erle Robotics provee distintos entornos para tres robots móviles (TurtleBot, ErleRover y ErleCopter), así como varios brazos robóticos. Recientemente se ha dejado de mantener esta extensión de OpenAI Gym para Ubuntu 16.04 y para ROS, tras el lanzamiento de Gym-Gazebo 2[9], que se ejecuta de forma nativa en ROS2 y únicamente ofrece la posibilidad de simular el robot modular MARA (*Modular Articulated Robotic Arm*).

[2]https://keras.io/
[3]https://www.microsoft.com/en-us/cognitive-toolkit/
[4]http://deeplearning.net/software/theano/
[5]https://gym.openai.com/
[6]https://github.com/erlerobot/gym-gazebo
[7]http://www.ros.org/
[8]http://gazebosim.org/
[9]https://github.com/AcutronicRobotics/gym-gazebo2

4.2. Entornos de Simulación

En este proyecto se han empleado varios entornos de simulación para evaluar los algoritmos de RL. Son entornos propios de Gym-Gazebo, versiones alteradas de estos o nuevos entornos creados para Gazebo e integrados en el *framework* de Gym-Gazebo.

En todos ellos, el robot TurtleBot[10], equipado con un láser 2D y una cámara, navega por el entorno generando distintas observaciones y cambiando de estado. El conjunto de acciones posibles está restringido a velocidad lineal positiva, velocidad angular positiva o velocidad angular negativa ($\dim \mathcal{A} = 3$):

$$\begin{aligned} a_1 &= \{v = 0.3, \quad \omega = 0\} \\ a_2 &= \{v = 0.05, \omega = 0.3\} \\ a_3 &= \{v = 0.05, \omega = -0.3\} \end{aligned}$$

Donde v es la velocidad lineal en $\mathrm{m\,s^{-1}}$ y ω la angular en $\mathrm{rad\,s^{-1}}$. El espacio de estados viene definido por las medidas del sensor láser por lo que su dimensión variará en función de la descripción del sensor. Además, el máximo número de pasos por episodio se ha limitado a 1000 para RL y 500 para DRL por su demanda computacional. Puesto que la exploración debe ocurrir sin colisiones con las paredes, el episodio también termina si el robot se acerca por debajo de un *threshold* de 0.2 m a cualquier obstáculo.

En las figuras 4.1 y 4.2 se muestran los tres entornos empleados, pudiendo destacar las similitudes entres el primero y el segundo y la complejidad del tercero.

Los procesos de entrenamiento y evaluación que se han llevado a cabo en ambas aproximaciones se muestran en la figura 4.3. En el caso de RL el módulo de memoria no es más que la *Q-table*, mientras que en el caso "*deep*" representa la ER de la que se extraen muestras para actualizar los parámetros de la red.

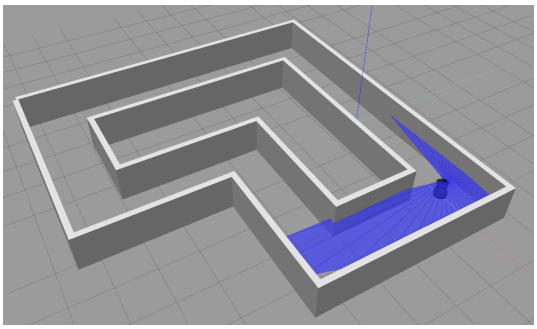
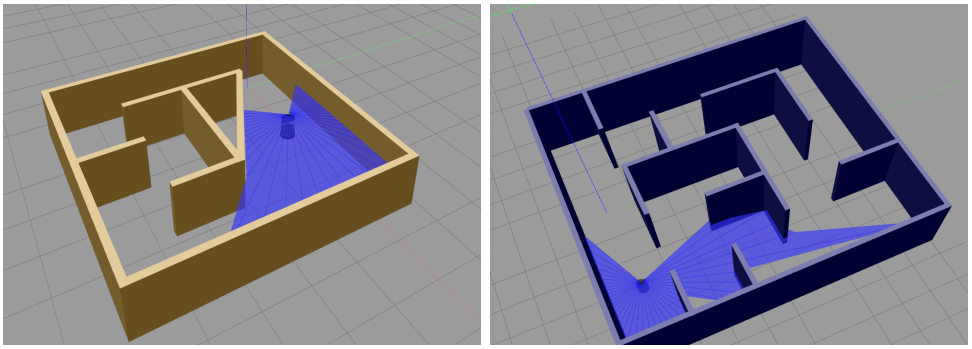


Figura 4.1: Escenario de entrenamiento.

[10] http://wiki.ros.org/Robots/TurtleBot



(a) Escenario 2. (b) Escenario 3.

Figura 4.2: Escenarios de generalización.

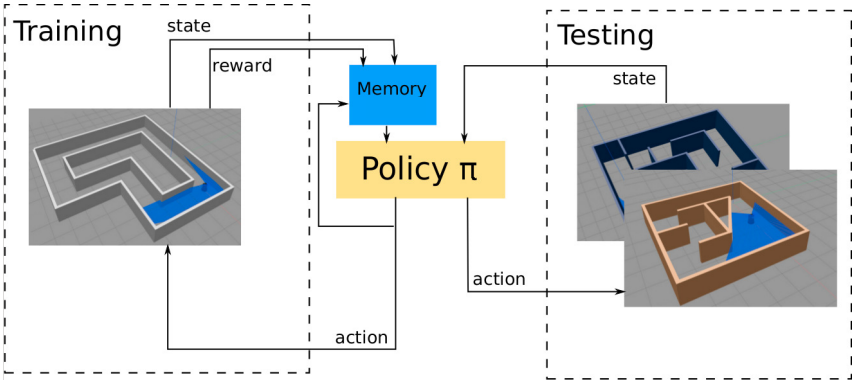


Figura 4.3: Procesos de entrenamiento y evaluación. Durante el primero la memoria guarda información de las recompensas obtenidas según el estado y la acción escogida, dando forma a la *policy*. En la fase de testeo, únicamente se evalúa la *policy*.

4.3. Aproximación RL: Q-Learning

4.3.1. *Traditional reward*

En primer lugar, se busca comprobar que las aproximaciones con DNN suponen una ventaja realmente sobre algoritmos de RL tradicionales. En la mayoría de trabajos que afirman capacidad de exploración se plantean recompensas extrínsecas clásicas, por lo que se ha realizado un experimento siguiendo estas premisas. Se ha entrenado un primer agente mediante *Q-learning* en el entorno 1 durante 10000 episodios ($\approx$ 125 horas en T1), con una *policy* ε*-greedy* con ε decreciente en el intervalo $[0.9, 0.05]$, factor de aprendizaje $\alpha = 0.2$, factor de descuento $\gamma = 0.8$ y la siguiente función recompensa:

$$R_t = \begin{cases} -\ 100 & \text{si hay colisión} \\ \ \ 0.5 & \text{si } \omega = 0 \\ -\ 0.05 & \text{si } \omega \neq 0 \end{cases} \tag{4.1}$$

Donde cada valor ha sido sintonizado de forma que se eviten giros y movimientos innecesarios motivados por mínimos locales (e.g. giros continuos).

Uno de los grandes retos en el problema de *Q-learning* es convertir el conjunto de lecturas del láser en estados para generar la *Q-table*. Esto se ha resuelto discretizando la lectura del láser en 5 datos equidistantes, pero otras opciones como calcular la media o el valor mínimo entre grupos de datos serían métodos también válidos.

En la figura 4.4 se muestran los resultados del aprendizaje. Se observa una curva de aprendizaje creciente que llega a estabilizarse, a pesar de haber parado el entrenamiento por su elevado coste computacional tras $\approx$ 7.5M *steps* y de cambiar los parámetros "recomendados" por los autores (Zamora et al., 2016). Nótese también el cambio de la tendencia conforme $\varepsilon \to 0$. Efectivamente el agente aprende a tomar decisiones favorables para navegar en el entorno evitando las paredes, consiguiendo alcanzar en muchas ocasiones el límite de pasos fijado por episodio. Sin embargo, surgen dos preguntas: ¿se trata realmente de una exploración, o simplemente está moviéndose para evitar los obstáculos como proponía Banerjee et al., (2018), i.e. navegando? y, ¿este aprendizaje puede extrapolarse o generalizarse a otros entornos distintos?.

A continuación se ha probado el agente entrenado en los entornos 2 y 3 durante 1000 episodios. En las figuras 4.5 y 4.6 se presentan los resultados obtenidos por el agente entrenado, un agente sin entrenar y un agente cuyas acciones están condicionadas por un factor de aleatoriedad elevado ($\varepsilon \approx 1$). En el entorno 2, a pesar de la variabilidad por la aparición de estados no contemplados en la *Q-table*, el comportamiento del agente entrenado (azul) es superior a los demás. En el entorno 3, completamente diferente y más complejo, sin embargo, los resultados son similares para los tres agentes. Se observa una ligera mejoría en cuanto a la recompensa o los pasos medios, pero en ningún caso se trata de una mejoría notable (recuérdese que han sido empleadas $\approx$ 125

horas de entrenamiento). La capacidad de generalización de este algoritmo puede considerarse limitada por la similitud entre los entornos y su complejdad, conclusión extendible a algoritmos similares de RL.

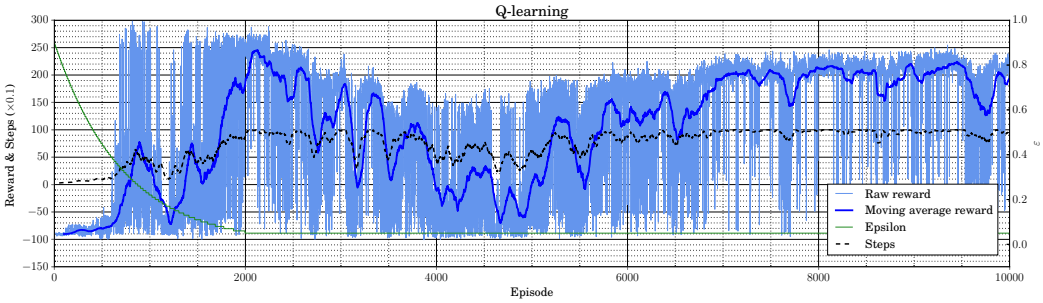
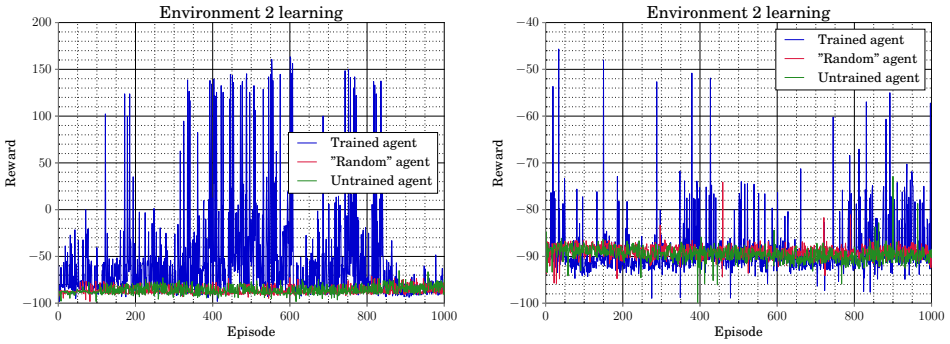
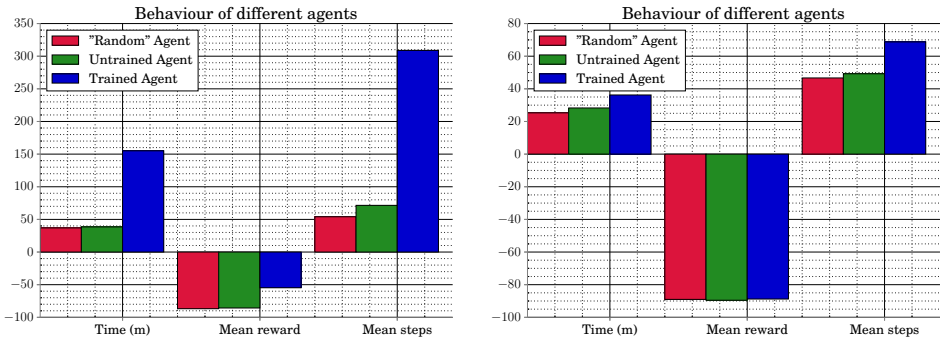


Figura 4.4: Recompensa durante el aprendizaje con *Q-learning* de un agente en el entorno 1 (azul claro), su media móvil (azul oscuro), los steps medios (negro) y ε (verde).



(a) Entorno 2 (b) Entorno 3

Figura 4.5: Recompensa de un agente estocástico (rojo), uno sin entrenar (verde) y el agente entrenado (azul) en los escenarios de generalización.



(a) Entorno 2 (b) Entorno 3

Figura 4.6: Comportamiento de los tres agentes mencionados en cuanto a tiempo de cómputo y recompensa y pasos medios por episodio.

4.3.2. *Uncertainty-based reward*

Considérese ahora la definición de exploración robótica (ver sección 2.1) en la que el objetivo es minimizar la incertidumbre de la localización del robot y el mapa creado. Una recompensa definida según la ecuación (4.1) únicamente está motivando la evasión de obstáculos y el movimiento por el centro de pasillos, y prueba de ello es la figura 4.7, donde se muestra la entropía en función del número de pasos (izquierda) y episodios (derecha) para el escenario 2. Además de la discontinuidad generada por revisitar estados conocidos en torno a los 200 pasos, se observa una tendencia desordenada de la entropía conforme aumentan los episodios, esto es, únicamente mejora o empeora arbitrariamente como consecuencia de intentar optimizar R_t.

Por el contrario, si se incluyese en la ec. (4.1) una componente inversa a una métrica de la matriz de covarianza (Σ), se motivaría visitar además aquellos estados que redujesen la incertidumbre y se evitarían otros indeseados que favoreciesen el aumento de la incertidumbre progresivamente, e.g. giros continuos en un punto o acercarse en exceso a las paredes. Se propone por tanto la siguiente función de recompensa aumentada:

$$R_u = \begin{cases} -\ 100 & \text{si hay colisión} \\ 0.5\ + \dfrac{1}{f(\Sigma)} & \text{si } \omega = 0 \\ -\ 0.05\ + \dfrac{1}{f(\Sigma)} & \text{si } \omega \neq 0 \end{cases} \tag{4.2}$$

Para disponer de una métrica de la covarianza durante la simulación se ha empleado el paquete de ROS *gmapping*[11] ya que es relativamente sencillo de usar y su demanda computacional no es alta. Tras su previa configuración y ajuste de parámetros, es capaz de ejecutar un algoritmo de SLAM basado

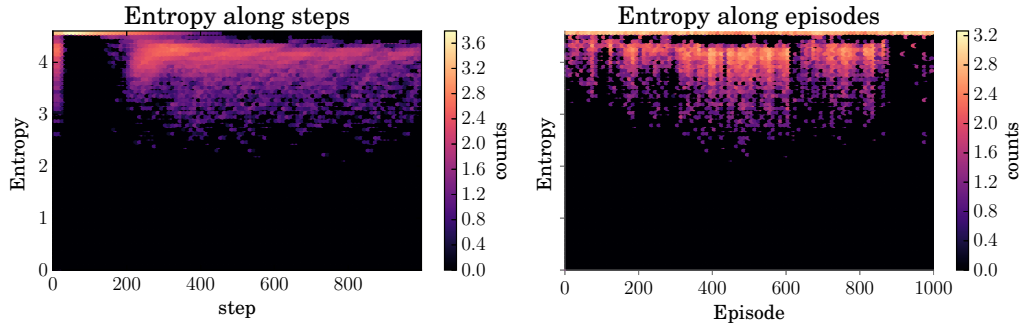


Figura 4.7: Evolución de la entropía en función de los episodios y steps en el escenario 2 con el agente entrenado con R_t. El mapa de color indica la ocurrencia logarítmica.

[11] http://wiki.ros.org/gmapping, https://openslam-org.github.io/gmapping.html

(a) Entorno 1 (b) Entorno 2 (c) Entorno 3

Figura 4.8: Visualización en RViz de los mapas creado mediante *gmapping* de los tres escenarios.

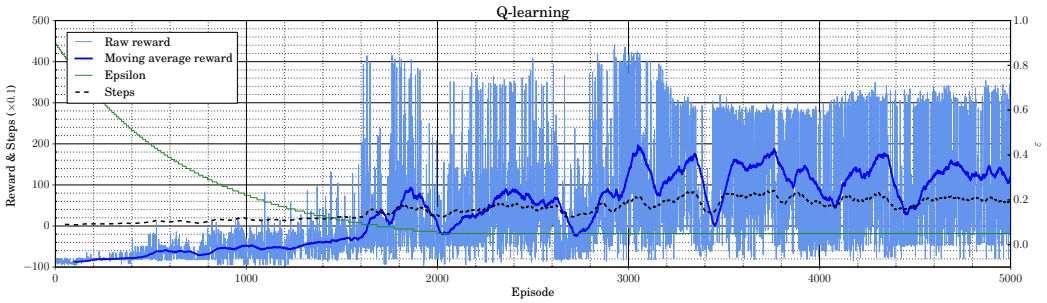


Figura 4.9: Recompensa durante el aprendizaje con *Q-learning* empleando R_u en el entorno 1 (azul claro), su media móvil (azul oscuro), los pasos medios (negro) y ε (verde).

en filtros de partículas de Rao-Blackwell (*Rao-Blackwellized Particle Filter*, RBPF) que calcula el mapa del entorno, la pose del robot en éste y una métrica de la incertidumbre: la entropía (i.e. $f(\Sigma) := \mathcal{H}$); a partir de la odometría y las medidas láser. En la figura 4.8 se presentan ejemplos de los mapas creados por el algoritmo, visualizados con RViz[12].

La figura 4.9 muestra el aprendizaje de un agente programado con la función de recompensa definida en la ecuación (4.2) para los 5000 primeros episodios ($\approx$ 2M *steps* y $\approx$ 29 horas en T1). Véase la notable diferencia entre las curvas de aprendizaje 4.4 y 4.9 en cuanto a tendencia, efecto de ε, estabilidad y convergencia, entre otros.

Finalmente, en la figura 4.10, se muestra la densidad de entropía con el agente entrenado con R_u. La evolución de $\mathcal{H}$ es ahora más ordenada y decreciente a lo largo de los episodios, dentro de lo esperado por la limitada capacidad de generalización del algoritmo al segundo entorno.

[12]http://wiki.ros.org/rviz

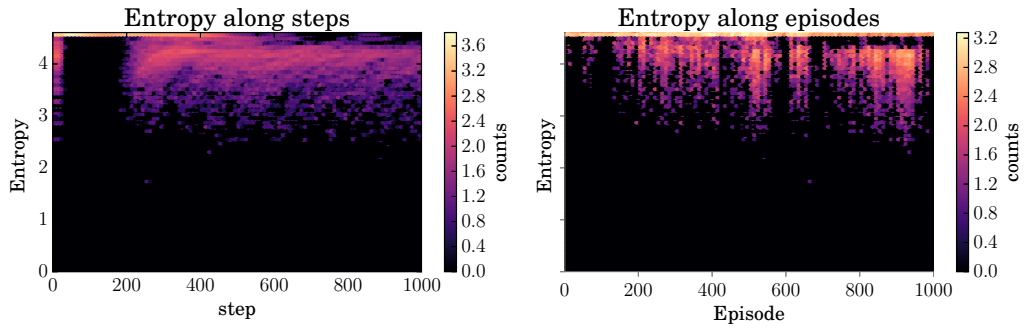


Figura 4.10: Evolución de la entropía en función de los episodios y steps en el escenario 2 con el agente entrenado con R_u. El mapa de color indica la ocurrencia logarítmica.

4.3.3. Conclusiones

Queda demostrada la incapacidad de algoritmos de (D)RL en general de explorar un entorno sin previamente incluir una componente adicional que motive el aprendizaje reduciendo la incertidumbre. Los experimentos con una componente inversa a la entropía conducen a resultados satisfactorios, motivando la reducción de incertidumbre incluso en entornos distintos del de aprendizaje. En ellos, sin embargo, el comportamiento del agente es mucho peor que en los mapas de entrenamiento, en función de su parecido y su complejidad. Además, es destacable la elevada necesidad computacional para resolver el problema.

4.4. Aproximación DRL: Deep Q-Learning

Analizado el marco de RL, se ha estudiado la aproximación con DNN, en la que la *Q-function* es sustituida por una red que actúa como *function approximator*. De esta forma quiere analizarse si las redes profundas aportan alguna ventaja a la hora de generalizar o tomar las decisiones repecto a la aproximación tradicional. A diferencia de los anteriores, estos experimentos se han ejecutado principalmente en la GPU de T2 ya que Keras (con TensorFlow *backend*) tiene soporte para ésta.

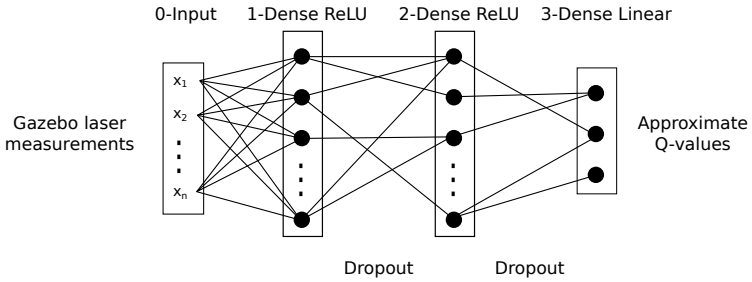
En primer lugar, han tenido que realizarse diversas modificaciones a la librería Gym-Gazebo para el correcto funcionamiento de la simulación con DNN. Entre las más representativas se encuentran: (i) la creación de un nuevo modelo de sensor láser con n rayos horizontales entre $\pm 180°$, que no tenga en cuenta todos los decimales del tipo `float32` ni los valores `NaN`, `+Inf` y `-Inf` ; (ii) una velocidad de ejecución de acciones que permita que el robot se mueva y Gazebo realice los cálculos necearios, esperando para ello varios milisegundos; (iii) la posibilidad de reaparecer en distintos puntos del mapa, reiniciando correctamente los nodos ROS de odometría, comando de velocidad y estado; (iv) el ajuste de los parámetros del *solver* de Gazebo para limitar el tiempo de simulación a 10 veces el tiempo real, permitiendo que los cálculos necesarios se ejecuten en todo momento; y (v) la nueva definición del modelo del robot para evitar desviaciones en la trayectoria y temblores no deseados, provocados por una descompensación inercial y por colisiones entre mallas.

Tras estas modificaciones, se han programado una *Deep Q-Network* (DQN), una *Double Deep Q-Network* (DDQN) y una *Dueling Double Deep Q-Network* (D3QN), que se presentan en el Anexo B. La función recompensa se ha definido de forma análoga a *Q-learning* como:

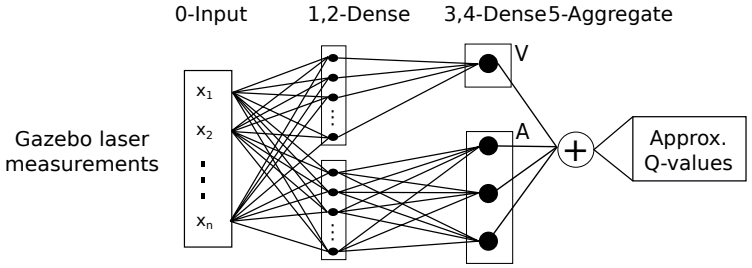
$$R_{DQN} = \begin{cases} -\ 100 & \text{si hay colisión} \\ \quad 1 & \text{si } \omega = 0 \\ -\ 0.05 & \text{si } \omega \neq 0 \end{cases} \tag{4.3}$$

La DQN, representada en la figura 4.11(a), está formada por una capa de entrada que recibe las mediciones del láser, 2 capas densas (24 y 12 *hidden units*) con activación ReLU, y una capa de salida densa con activación lineal (de tamaño igual al número de acciones posibles). Las capas intermedias están regularizadas mediante *dropout*. El método de optimización es RMSProp con *learning rate* $\alpha = 2.5\text{e}{-4}$ y como función de pérdida se usa el error cuadrático medio. La red emplea ER con un *buffer* de 50000 vectores de la forma $(s_t, a_t, r_t, s_{t+1}, e_t)$, del que se muestrean lotes arbitrarios de tamaño 64 (donde e_t indica si se trata del último estado de un episodio). Se sigue además una *policy* $\varepsilon - greedy$ con ε decreciente en el intervalo $[1, 0.05]$ durante los primeros episodios. Los parámetros de las redes principal, θ, y objetivo, θ^*, se sincronizan cada 10000 *steps* para disponer de un objetivo estacionario (*hard updates*), y se inicializan según una distribución uniforme (LeCun). Para esta

configuración, el número de parámetros entrenables es $\theta \approx 600$. La tabla 4.1 muestra un resumen de los parámetros descritos. Por su parte, el módulo de memoria se ha programado sobre el existente de Gym-Gazebo, empleando la estructura `deque()` de alto rendimiento para extracción y guardado de tuplas en Python. La red D3QN, figura 4.11(b), comparte la mayoría de parámetros con la primera, habiendose modificado principalmente la arquitectura y el módulo de memoria (ver Anexo B). Para conseguir resultados repetibles, se han fijado las semillas de aleatoriedad de todas las librerías empleadas, aunque un pequeño grado de variabilidad existe debido a la simulación en Gazebo.



(a) DQN con *dropout*



(b) *Dueling* DDQN

Figura 4.11: Arquitectura de las redes empleadas, donde los circulos negros representan las neuronas de la red.

Parámetro	**Símbolo**	**Valor**
Factor de aprendizaje	α	0.00025
Factor de descuento	γ	0.99
Factor de dropout capa 1	d_1	0.5
Factor de dropout capa 2	d_2	0
Tamaño de ER	$\mathcal{M}$	50000
Tamaño de muestreo	b	64
Núm. entradas	n_{in}	10
Núm. salidas	n_{out}	3
Steps de exploración	U_e	100
Steps entre actualizaciones	U	10000

Tabla 4.1: Resumen de los parámetros de aprendizaje de la DQN.

4.4.1. Análisis de resultados

Se han realizado numerosas simulaciones con distintas combinaciones de parámetros y arquitecturas, detectando una enorme sensibilidad no sólo a la variación de éstos sino también a su relación. Factor de aprendizaje y tamaño de muestreo de la memoria son claves en la simulación, siendo el efecto de disminuir el primero similar a aumentar el segundo (Smith et al., 2017). *Learning rates* menores del propuesto conducen a aprendizajes extremadamente lentos, mientras que mayores de 0.001 provocan inestabilidades. El tamaño del muestreo mínimo parece estar en torno a 64 muestras para conseguir aprender a una velocidad aceptable, mientras que puede ser aumentado progresivamente durante el entrenamiento hasta valores próximos a 300 sin penalización computacional elevada. El tamaño de la memoria, por su parte, parece no tener un efecto notable siempre que se encuentre en valores mínimos razonables que permitan una cierta variedad en las experiencias contenidas (e.g. 10-50k). Tamaños de hasta 500k se han evaluado, no presentando mejora en la *policy* conseguida. La arquitectura de la red es otro de los factores críticos: un gran número de entradas y/o de posibles acciones hace que el aprendizaje se ralentice mucho y consigue únicamente que la red termine ignorando parte de éstas, como puede observarse en la figura 4.12, donde conforme avanza el entrenamiento cada vez más entradas se ignoran; mientras que un bajo número de lecturas de láser hace difícil el reconocimiento de patrones similares. Aumentar el número de neuronas o capas por encima del propuesto ralentiza también la convergencia sin una mejora significativa en el comportamiento del robot en simulaciones cortas, aunque su efecto en generalización y en simulaciones más largas sería beneficioso. Por otra parte, la actualización de pesos de la red objetivo muestra también una variación significativa en los resultados. Si los *Q-targets* no están fijos durante un número de episodios que permitan la convergencia a éstos, se producen oscilaciones e inestabilidades, como ya se habia visto en (S. Chen, 2018). Actualizaciones continuas (*soft updates*) de la *target network* también se han evaluado (e.g. interpolación $\tau \times \theta^* + (1 - \tau) \times \theta$ con $\tau \in (0.001, 0.005)$), pero no se ha visto una mejoría notable para el coste computacional que supone actualizar los pesos cada *step*. El valor de velocidad lineal y angular son también críticos y deben ser ajustados en función de las dimensiones del entorno. Finalmente, un parámetro que ha demostrado ser de una importancia y sensibilidad abismal es la función de recompensa. El diseño de ésta en cuanto a valores absolutos y relación entre ellos tiene efecto, entre otros, en: convergencia a *policies* no óptimas, velocidad de aprendizaje y error de estimación de *Q-values* (i.e. *loss function*), y por tanto en problemas derivados como *exploding gradients*. Pese a que recompensas contenidas en el intervalo $[-1, 1]$ (*reward clipping*) consiguen funciones de pérdida y gradientes acotados y menos bruscos, no ha resultado en una convergencia a mejores *policies* en los horizontes temporales contemplados.

Como puede comprobarse en (Mohaimenian Pour et al., 2017), donde se entrena un agente similar con parámetros similares a los propuestos, la conver-

gencia a una *policy* óptima es conseguible tras un periodo de entrenamiento suficientemente grande. En ese caso, tras varios días de entrenamiento en un entorno más simple y empleando una GPU Nvidia Titan Xp. Por este motivo, la convergencia a una *policy* óptima se descarta en los experimentos realizados. Un ejemplo de aprendizaje de una *policy* exitosa se muestra en la figura 4.13, con la que el agente es capaz de recorrer el entorno completo en aproximadamente el 85 % de ocasiones. En este caso, 100 son las entradas de la red y 21 sus salidas, al igual que en (Mohaimenian Pour et al., 2017), aunque los valores absolutos de la función de recompensa son ligeramente distintos y la red contiene un número de neuronas inferior. Como puede observarse, una regla subóptima se sigue durante casi la mitad del entrenamiento, traducida en un agente que únicamente sabe avanzar en línea recta y girar en un único sentido cuando va a chocarse. No se ha detectado qué hace realmente al agente salir de dicha *policy* pero supone un aprendizaje casi repentino de cómo girar en ambas direcciones. En entornos distintos al de aprendizaje, tanto esta red como la propuesta por Mohaimenian se comportan de forma muy inferior, mostrando comportamientos mejores que los de un agente aleatorio únicamente si los entornos tienen una estructura similar (e.g. pasillos de igual tamaño, giros similares...). Nótese que el entrenamiento ha necesitado un total de $\approx$ 120 horas para los $\approx$ 3.1M de *steps*.

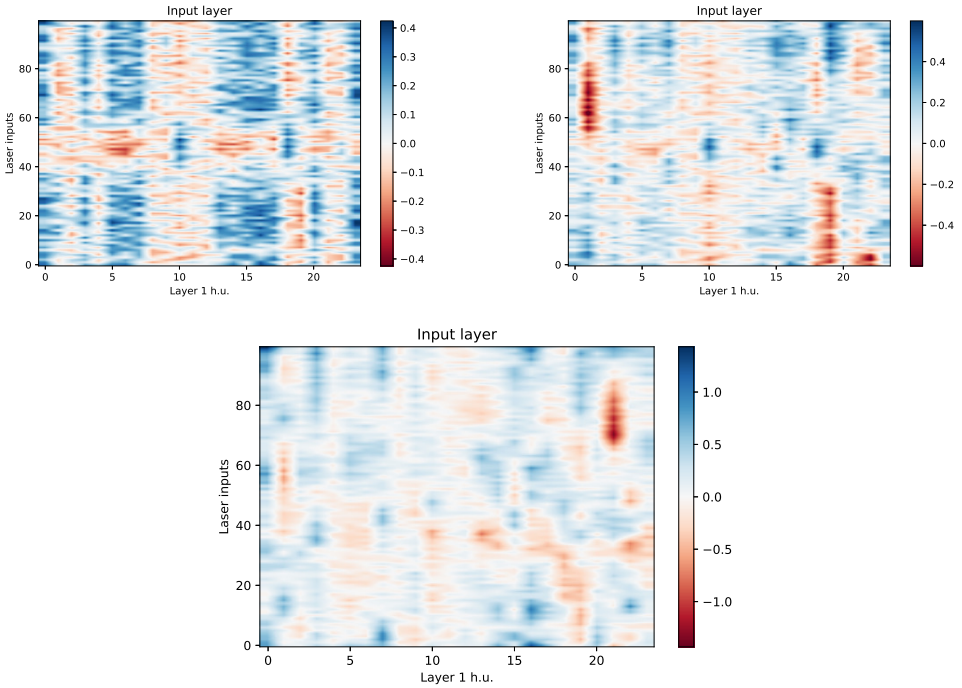


Figura 4.12: Evolución de los parámetros de la primera capa densa de una red cuya entrada son 100 lecturas de láser, tras 100, 1000 y 4000 episodios de entrenamiento. Nótese la especialización de algunas neuronas en zonas del láser o cómo muchas entradas son prácticamente ignoradas conforme se entrena la red.

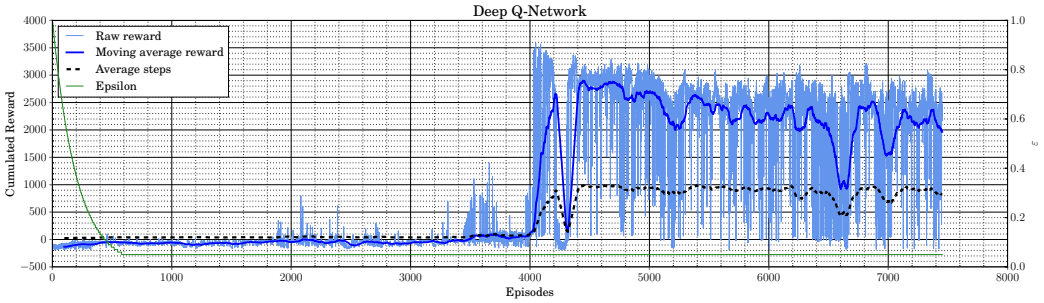
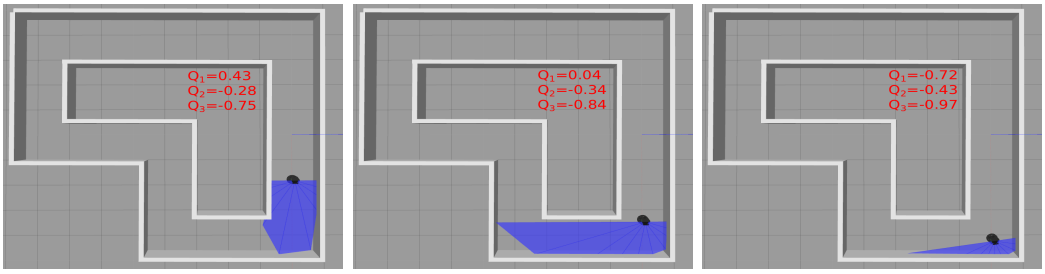


Figura 4.13: Recompensa durante el aprendizaje de DQN en el escenario 1 (azul claro), su media móvil (azul oscuro), ε (verde) y los *steps* (negro). Máximos *steps*: 1000, recompensa teórica estimada máxima: ≈ 4000.

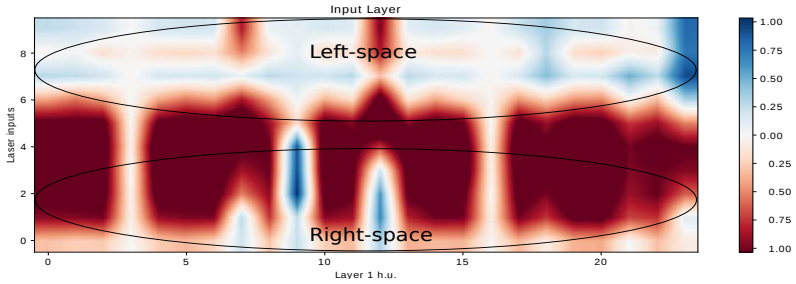
Las figuras 4.14(b) y 4.14(c) contienen los *kernels* de entrada y salida de la red como ejemplo de una mala *policy* (similar a la del agente anterior durante la primera parte de la simulación) a la que otro agente entrenado converge tras más de 3000 episodios de entrenamiento con la configuración de la tabla 4.1 y la función de recompensa de la ec. (4.3). En ella, los pesos se configuran de forma que las lecturas del láser a la derecha del agente no se ponderan bien y terminan en *Q-values* subestimados. Recuérdese que valores negativos son ignorados con activaciones ReLU, como puede verse en la figura 4.15: cuando las lecturas láser centrales y a la izquierda no son suficientemente grandes para mitigar el efecto negativo de las lecturas a la derecha, resulta en una gran cantidad de activaciones nulas (*dying ReLU*). En la figura 4.14(a) se puede osbervar el comportamiento del agente en la simulación y los *Q-values* estimados en una situación similar a la de la figura 4.15.

Una solución a este problema podría encontrarse en usar funciones de activación diferentes. Se ha entrenado una red similar, incluyendo 24 neuronas en ambas capas densas y 100 entradas láser, pero con activación LeakyReLU ($\theta \approx 3000$). Esta función no genera salida nula ante valores negativos sino un pequeño valor negativo función de la entrada. En la figura 4.16 se muestran los resultados de entrenamiento de un agente con regularización *dropout* tras la primera capa[13] (azul) y sin ella (rojo) tras 700 episodios ($\approx$ 28 horas). En primer lugar, se observa que el agente sin *dropout* converge a una *policy* cuasi-óptima durante el entenamiento, que tiene además una tendencia creciente. El agente con *dropout* converge a una buena *policy* tras 300 episodios que luego "olvida" repentinamente y parece "reaprender" más adelante. El uso de *dropout* introduce una varianza adicional al problema de redes neuronales que en el caso de SL se busca para evitar *overfitting*, pero que en DRL genera inestabilidades en el aprendizaje, puesto que la selección de acciones ε-greedy, el objetivo móvil y el propio entorno ya son una fuente de varianza enorme. La posible ventaja de solucionar el *overfitting* se ve nublada por la gran cantidad de neuronas muertas que hacen oscilar el aprendizaje, necesitando un tiempo

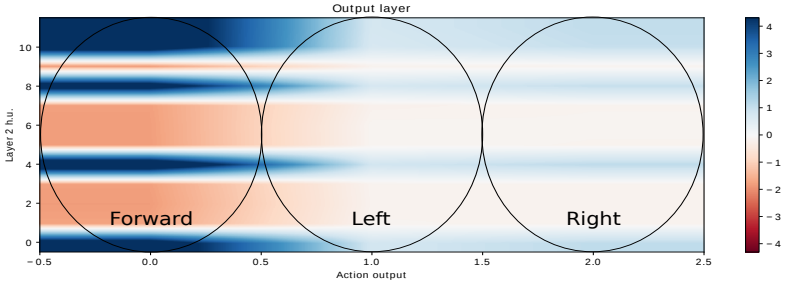
[13] También se ha analizado el comportamiento de un agente con *dropout* tras la segunda capa, siendo sus resultados similares.



(a) Secuencia de Gazebo de acercamiento a una esquina, y *Q-values*.



(b) Kernel de entrada.



(c) Kernel de salida.

Figura 4.14: Ejemplo de mala *policy*, donde el agente no considera giros a la derecha.

```
[2.536 0.979 0.653 0.538 0.511 0.55  0.685 0.735 0.652 0.657]   <- Inputs
Tensor("dense_1/BiasAdd:0", shape=(?, 24), dtype=float32)
[[-3.509138   -3.3802967  -3.770054   -0.09738671 -3.7229202  -4.1320806
  -3.765582    0.16527224 -1.7223551   1.5927334  -3.7156594  -1.8727303
   1.4738802  -1.9286959  -3.9608574  -3.8000574  -0.21094428 -4.0315146
  -2.0569663  -4.484127   -4.4910707  -1.6346238  -1.7618546   4.4096932 ]]
Tensor("activation_1/Relu:0", shape=(?, 24), dtype=float32)
[[0.         0.         0.         0.         0.         0.
  0.         0.16527224 0.         1.5927334  0.         0.
  1.4738802  0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         4.4096932 ]]
Tensor("dropout_1/cond/Merge:0", shape=(?, 24), dtype=float32)
[[0.         0.         0.         0.         0.         0.
  0.         0.16527224 0.         1.5927334  0.         0.
  1.4738802  0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         4.4096932 ]]
Tensor("dense_2/BiasAdd:0", shape=(?, 12), dtype=float32)
[[ 0.01666395 -0.92911524 -0.7790813  -0.87761927  0.01952073 -1.1484225
  -0.96237314 -1.0454454   0.02194455 -0.9766209   0.02511705  0.01496975]]
Tensor("activation_2/Relu:0", shape=(?, 12), dtype=float32)
[[0.01666395 0.         0.         0.         0.01952073 0.
  0.         0.         0.02194455 0.         0.02511705 0.01496975]]
Tensor("dense_3/BiasAdd:0", shape=(?, 3), dtype=float32)
[[-0.17743045 -0.36622384 -0.8733024 ]]   <- Q-values
[-0.17743045 -0.36622384 -0.8733024 ]
```

Figura 4.15: Salida de las capas de la red en la simulación mostrada en la figura previa, donde puede verse la gran cantidad de elementos nulos.

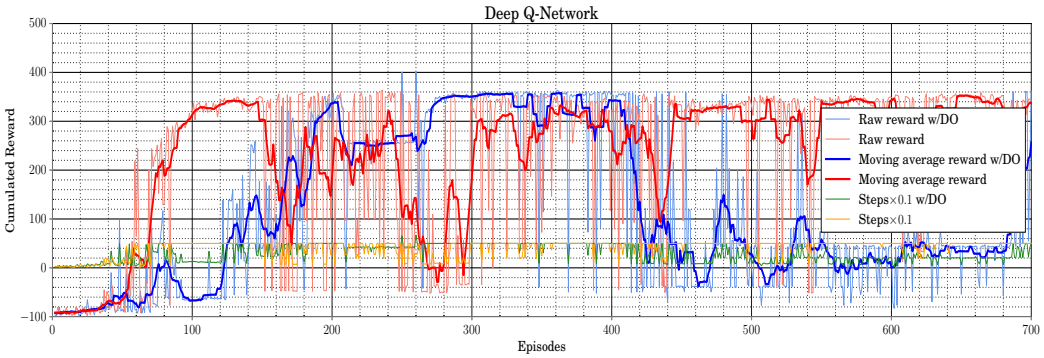


Figura 4.16: Recompensa acumulada durante el aprendizaje de DQN con LeakyReLU en el escenario 1 y su media móvil con (azul) y sin (rojo) *dropout*; y los *steps* en ambos casos (verde y amarillo, respectivamente).

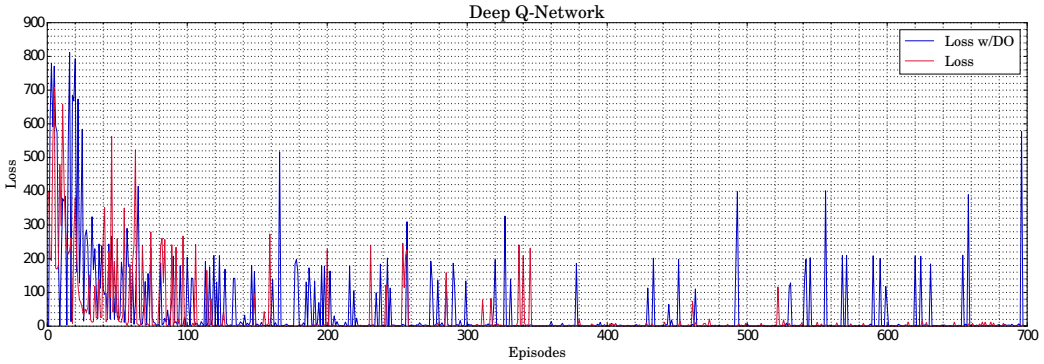


Figura 4.17: Función de pérdida durante el aprendizaje de DQN con LeakyReLU en el escenario con (azul) y sin (rojo) *dropout*.

de aprendizaje mucho más elevado para estabilizarse. La figura 4.17 muestra las funciones de pérdida para ambos agentes a lo largo del entrenamiento, respaldando estas conclusiones.

Ambos agentes se han evaluado en los tres entornos durante 50 episodios con una buena *policy* del entrenamiento (300 episodios con *dropout* y 700 episodios sin *dropout*), en los que ni se guarda información en la memoria ni se modifican los pesos. Cada evaluación se ha repetido 5 veces (3 para el primer escenario) liberando las semillas de aleatoriedad. En la tabla 4.2 se presentan el ratio de éxito (*success ratio*, SR) o porcentaje de intentos en los que se logra explorar el entorno completo al menos una vez, los pasos medios, la recompensa media acumulada en cada episodio, la recompensa teórica máxima estimada y la que obtendría una persona (sin chocarse); así como las desviaciones estándar de estas métricas entre paréntesis. Ambos agentes consiguen resolver perfectamente el entorno de entrenamiento, y aunque lo hacen con una *policy* subóptima, la recompensa es ligeramente superior a la que obtendría una persona. El segundo escenario es resuelto en el 91 % de ocasiones por el agente con *dropout* y en el 72 % por el que no tiene; aunque por el número de pasos y

la recompensa media se puede ver que ambos realizan numerosos giros innecesarios. Puede observarse cómo el *dropout* favorece la generalización, pese a que la fase de entrenamiento fue más inestable, ya que el agente está aprendiendo características "más generales" del entorno (Farebrother et al., 2018). El tercer escenario muestra los peores resultados, no consiguiéndose la exploración completa en ninguna ocasión por la dificultad que suponen los caminos múltiples o las zonas sin salida. Se nota una gran falta de entrenamiento para la resolución de este escenario y cómo las características aprendidas por el agente con *dropout* no mejoran el comportamiento del agente.

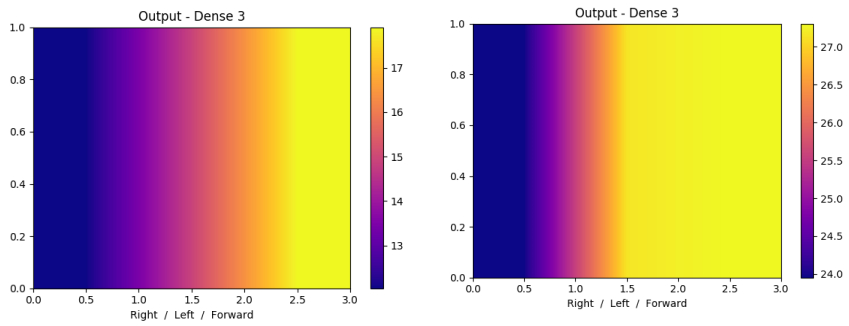
	Do (%)	**SR(%)**	**$\overline{\text{Steps}}$**	**$\overline{\text{R}}$**	**R teórica**	**R humano**
Entorno 1	50	100 (0)	500 (0)	350.52 (1)	$\lesssim 425$	≈ 345
	0	100 (0)	500 (0)	352.14 (0.1)		
Entorno 2	50	91.5 (4.1)	404 (16)	176.32(13.7)	$\lesssim 350$	≈ 300
	0	72.8 (6.4)	312 (13)	95.55 (14.4)		
Entorno 3	50	0 (0)	103(0.5)	-86.52 (0.6)	$\lesssim 325$	≈ 300
	0	0 (0)	170 (15)	-24.89 (8.3)		

Tabla 4.2: Resultados de evaluación del agente DQN con LeakyReLU, con y sin *dropout* en los tres entornos.

Las figuras 4.19 a 4.22 contienen la evolución de la información a través de las dos redes entrenadas en cuatro instantes de un episodio de evaluación del primer entorno: dos en los que la mejor opción es continuar recto y dos donde es mejor girar. En cada figura se muesrta el agente en Gazebo, seguido de cuatro gráficos para cada red. El primero de ellos (arriba izquierda) contiene la lectura del sensor láser (linea negra y primer mapa de color) y en qué lecturas del sensor la red está prestando más atención, i.e. contribuye más a la salida de la red (*saliency map*, segundo mapa de color). Tras este, se presentan la salida de la primera capa densa de la red (abajo izquierda), de la segunda (arriba derecha) y de la tercera (abajo derecha). Este último mapa corresponde al *Q-value* de cada acción, y por ende, a la acción que se escogerá con probabilidad $1 - \varepsilon$, siendo el valor contenido en $[0, 1]$ correspondiente a girar a la derecha, $(1, 2]$ a girar a la izquierda y $(2, 3]$ a continuar recto. Nótese que en todos los mapas cuanto más oscuro es el color más pequeño es el valor que representa.

En primer lugar, cabe destacar que en el caso con *dropout* son muchas menos las entradas significativas, puesto que se está obviando información. Es recurrente también que entradas centrales y laterales sean más empleadas, tendiendo a "ignorar" grupos intermedios y evidenciando un excesivo número de entradas. En los mapas de color de las tres capas, destaca especialmente la escala: la red con *dropout* tiene un rango de valores mucho menor, haciendo especial uso de valores negativos. En la red sin regularización la contribución de activaciones positivas es mucho mayor que las negativas y en pocos casos se propagan valores negativos hasta la capa de salida. Podría entenderse entonces que la red con *dropout* se ha entrenado de forma que las mediciones más bajas son las más ponderadas (negativamente) y por tanto se descartan, mientras

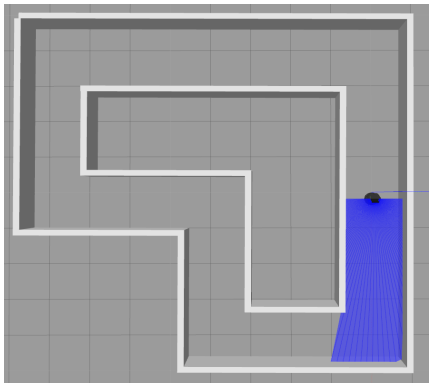
que en la otra red son las más altas las que se ponderan más y generan un *Q-value* mayor para la acción que conduce a esa zona. La red sin *dropout* tiene una tendencia clara en todos los casos, tanto en la segunda como en la tercera capa: unas cuantas neuronas son las principales y la ligera modificación de las demás deriva en una acción u otra. A pesar de que la capa final dista bastante tanto cualitativa como cuantitativamente, en todos los ejemplos se escoge la misma acción. Nótese también que un mapa de color más difuminado en la última capa estaría indicando menor seguridad en la acción escogida. En la figura 4.18 se muestra un mismo instante al comienzo de la evaluación donde la acción preferida es ir recto para dos agentes con la configuración de la red sin *dropout* tras 300 y 700 episodios de entrenamiento. Al avanzar el entrenamiento la seguridad de tomar la tercera acción es mayor y por tanto el mapa de color está menos difuminado. Este hecho tiene especial importancia en estas redes entrenadas durante poco tiempo, puesto que en numerosas ocasiones durante la evaluación el agente "duda" entre dos de las acciones y termina concurriendo en *policies* subóptimas.



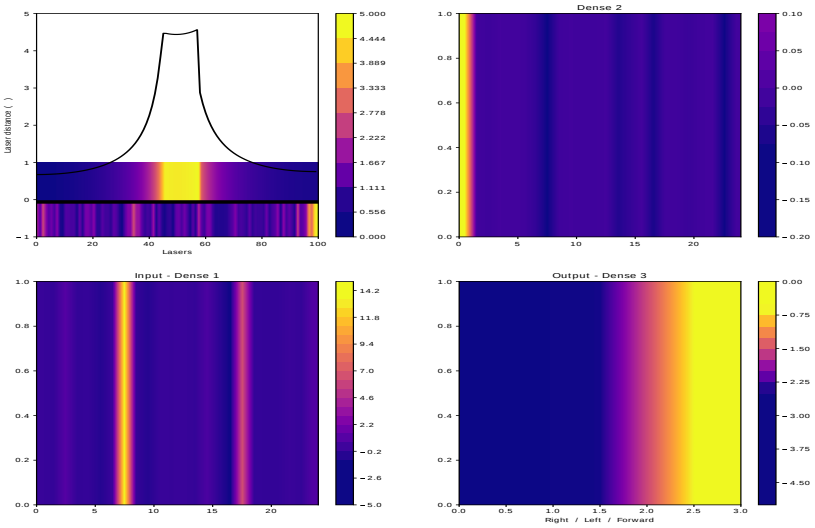
(a) (b)

Figura 4.18: *Q-values* estimados en el instante inicial tras (a) 300 y (b) 700 episodios de entrenamiento.

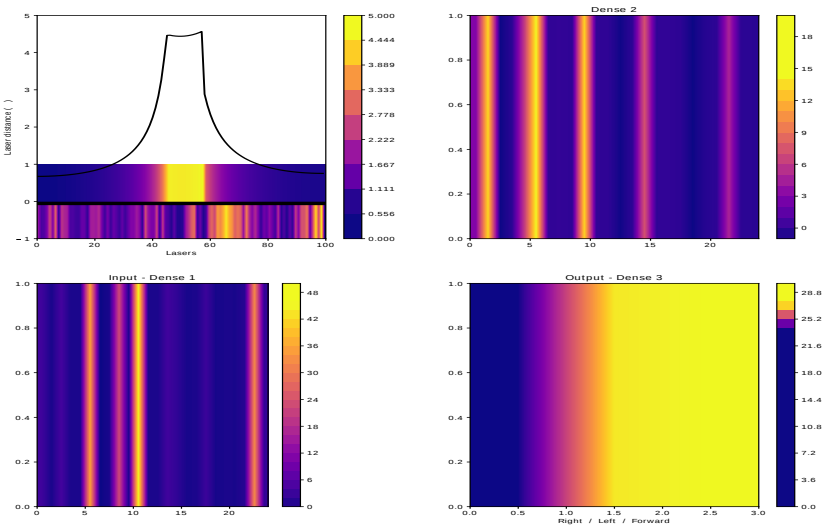
Finalmente, se muestra una comparación entre la red sin *dropout* y sus análogas *Double* DQN y *Double Dueling* DQN (ver Anexo B) en 500 episodios de aprendizaje ($\approx 15 - 20$ horas). La figura 4.23 contiene las tres curvas de aprendizaje, donde se muestra la recompensa acumulada y su media móvil con *outlier removal*. Se observa que la DDQN tiene un aprendizaje mucho más estable que la DQN y converge ligeramente más rápido, aunque las recompensas máximas conseguidas son sutilmente inferiores. La red D3QN, en la que una inicialización de pesos no aleatoria ha sido necesaria, está formada por dos flujos paralelos: el primero de ellos contiene una capa de extracción de *features* de 24 neuronas y activación LeakyReLU y una capa densa con una neurona que codifica $V(s)$, mientras que el segundo tiene otra capa de extracción análoga, una capa densa de 3 neuronas que codifica $A(s, a)$ y una capa que calcula el segundo término de la ecuación (B.6). Ambos flujos se unen en una capa de agregación final que codifica $Q(s, a)$ (ver figura 4.11(b), $\theta \approx 5000$). Puede notarse que el aprendizaje de esta red, que además emplea *Prioritised Experience*



(a) Instante 1.

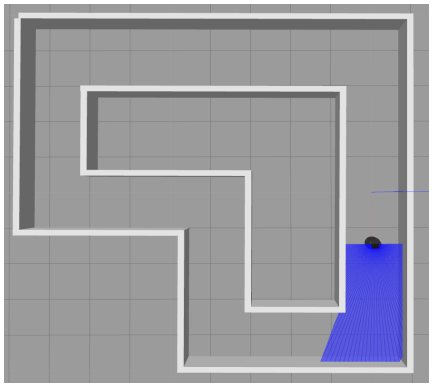


(b) Visualización de la red con *dropout* para el instante 1.

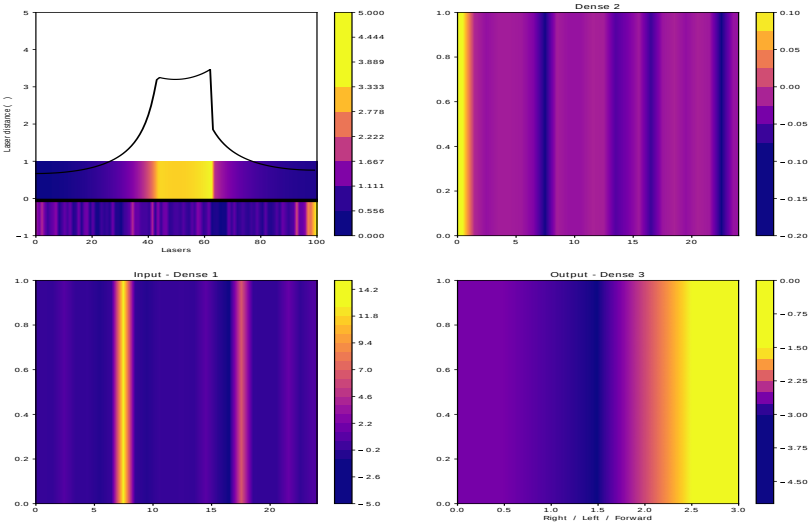


(c) Visualización de la red sin *dropout* para el instante 1.

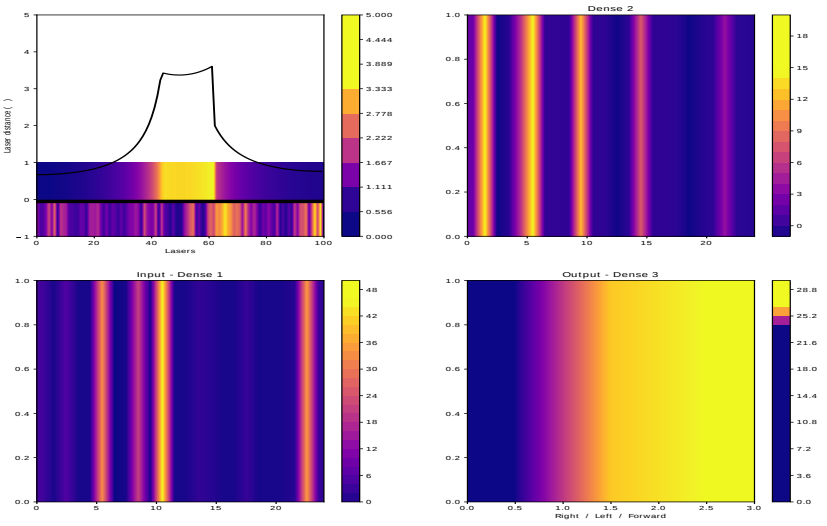
Figura 4.19: Evolución de la información a través de la red neuronal en el primer tramo.



(a) Instante 2.

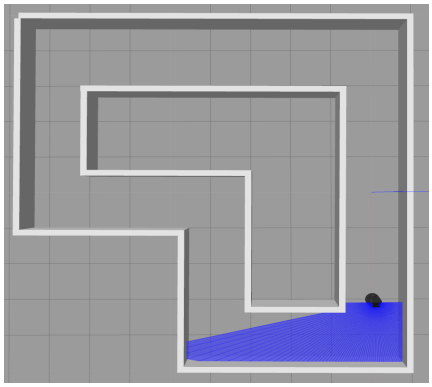


(b) Visualización de la red con *dropout* para el instante 2.

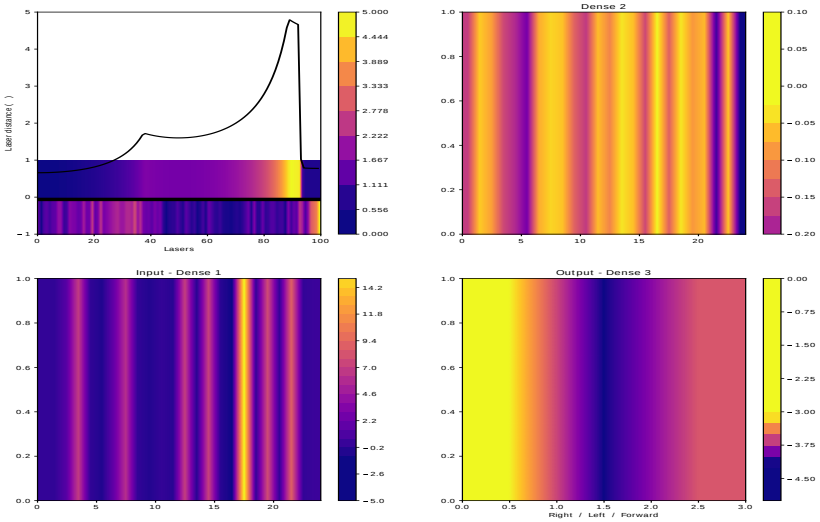


(c) Visualización de la red sin *dropout* para el instante 2.

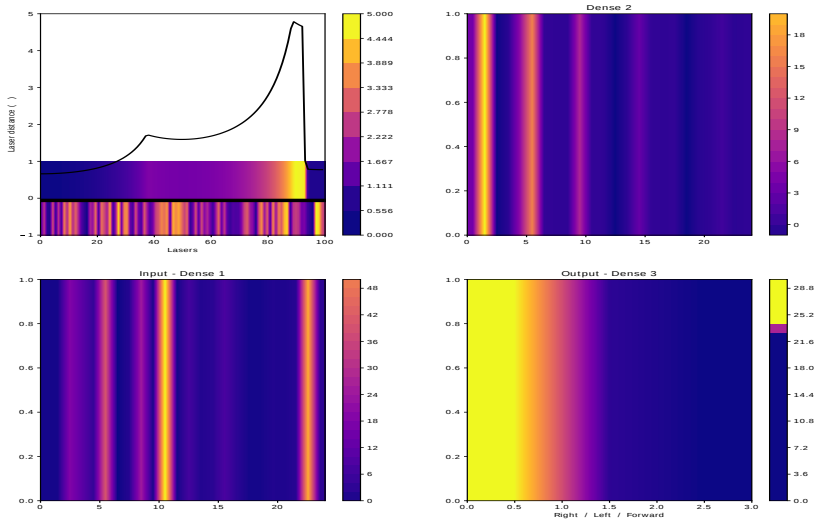
Figura 4.20: Evolución de la información a través de la red neuronal en el segundo tramo.



(a) Instante 3.

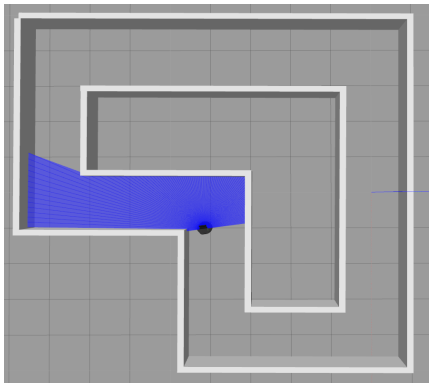


(b) Visualización de la red con *dropout* para el instante 3.

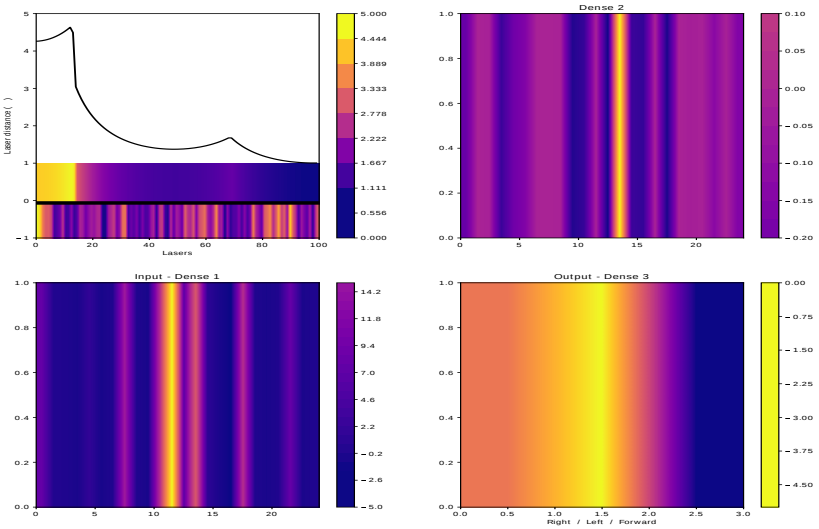


(c) Visualización de la red sin *dropout* para el instante 3.

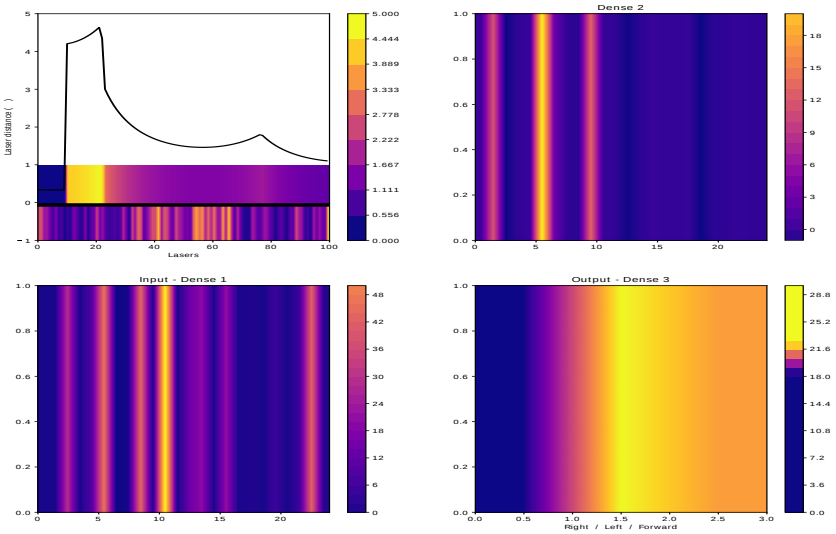
Figura 4.21: Evolución de la información a través de la red neuronal en el tercer tramo.



(a) Instante 4.



(b) Visualización de la red con *dropout* para el instante 4.



(c) Visualización de la red sin *dropout* para el instante 4.

Figura 4.22: Evolución de la información a través de la red neuronal en el cuarto tramo.

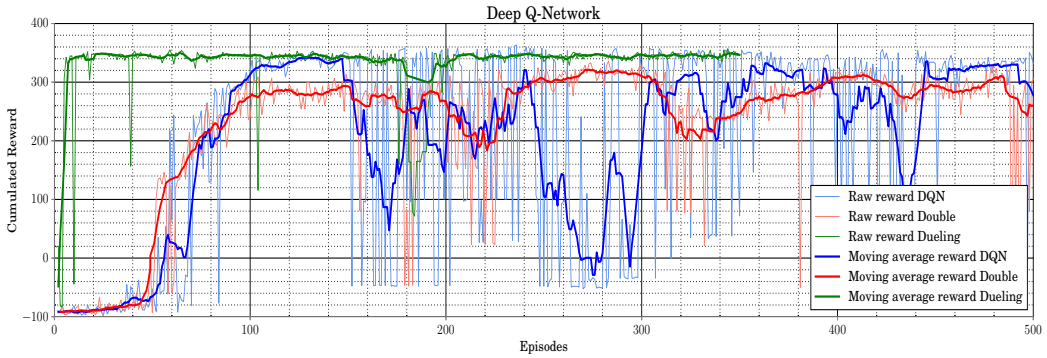


Figura 4.23: Recompensa acumulada durante el aprendizaje en el primer entorno de DQN (azul), DDQN (rojo) y D3QN (verde). Se muestran los datos en bruto y su media móvil sin *outliers*.

Replay (PER), es notablemente superior a las anteriores tanto en términos de estabilidad como valor absoluto. Esta simulación se ha realizado en un menor número de episodios por la demanda computacional, siendo el tiempo de ejecución similar a las anteriores. Mediante un aprendizaje previo, se ha permitido converger los pesos a una buena *policy*, inicializando con ellos la red y estando así mucho más cerca de la solución del problema de optimización (ver zona inicial de aprendizaje).

La tabla 4.3 muestra los resultados de generalización de las tres redes con las mejores *policies* escogidas de cada una de ellas (700, 300 y 300 episodios, respectivamente). La DDQN se comporta mucho mejor en el segundo escenario que su predecesora, logrando un 100 % de éxito y duplicando la recompensa media obtenida. En el primer y tercer escenarios, el comportamiento es similares entre ambas redes, teniendo la red *double* recompensas ligeramente inferiores, como ya se apreciaba en el aprendizaje. La D3QN, por su parte, consigue sobrepasar a ambas redes en el primer escenario. En el segundo escenario, el ratio de éxito baja al 90 %, pero como puede observarse, la recompensa media es incluso superior a la del agente DDQN (que conseguía un ratio del 100 %). Esto es debido a que aunque en algunas ocasiones el agente D3QN se choca, cuando no lo hace la recompensa obtenida es muy superior a la de DDQN: ≈ 267 frente a ≈ 192. Esta elevada recompensa en caso de completar el episodio con 500 pasos ya se veía con DQN, pero en este caso, ocurría un número de veces muy bajo. Finalmente, en el tercer escenario, el agente recorre los 500 *steps* en la mayoría de ocasiones, con una buena *policy*, generando así resultados muy superiores a los anteriores. De nuevo, algunas zonas del mapa no se exploran, por lo que el ratio de éxito es del 0 %. Cabe remarcar que en esta red solo hay una capa que codifique las mediciones de cada flujo, mientras que en las anteriores, la informacion pasaba por dos capas consecutivas. [14]

[14]Nótese que el agente DQN está empleando parámetros entrenados tras 700 episodios, mientras que DDQN y D3QN lo hacen tras 300 episodios, por lo que la "seguridad" en cada decisión, y por ende el ratio de éxito, se ven afectados.

	Arquitectura	SR(%)	$\overline{\text{Steps}}$	$\overline{\text{R}}$
Entorno 1	DQN	100 (0)	500 (0)	352.14 (0.1)
	Double DQN	100 (0)	500 (0)	337.56 (2.2)
	Double Dueling DQN	100 (0)	500 (0)	355.31 (0.3)
Entorno 2	DQN	72.8 (6.4)	312 (13)	95.55 (14.4)
	Double DQN	100 (0)	500 (0)	192.52 (3.0)
	Double Dueling DQN	89.3 (1.1)	419(9.3)	196.5 (5.7)
	DQN & warm-up	100 (0)	500 (0)	269.98 (4.4)
Entorno 3	DQN	0 (0)	170 (15)	-24.89 (8.3)
	Double DQN	0 (0)	253 (16)	-42.15 (4.8)
	Double Dueling DQN	0 (0)	432 (18)	241.56(16.5)

Tabla 4.3: Resultados de evaluación de los agentes DQN, DDQN y D3QN en los tres entornos.

Para la DQN se ha realizado además otro experimento en el que se permite un breve periodo de aprendizaje antes de realizar la evaluación en el segundo entorno (*warm-up*), de forma que la red adapte sus pesos ligeramente. En la figura 4.24 se muestran los resultados de evaluación cada 10 episodios de pre-entrenamiento; empleando un *set-up* igual al de entrenamientos anteriores pero acotando los gradientes en RMSProp para evitar cambios bruscos en los pesos. Nótese que tras comenzar el entrenamiento, la *policy* empeora notablemente, ya que comienza a adaptar los parámetros, pero tras un periodo de tiempo razonable, mejora hasta superar a redes con arquitecturas más complejas. En el tercer escenario se ha detectado que el periodo de pre-entrenamiento necesario es excesivamente elevado para apreciar mejoría, por lo que no se muestran los resultados.

4.4.2. Conclusiones

La aproximación con redes neuronales profundas ha demostrado tener una gran complejidad en el ajuste de parámetros con respecto al RL tradicional. La convergencia a *policies* óptimas o cuasi-óptimas es difícil, y más en entornos de simulación como Gazebo, donde el entorno se representa de forma detallada y sus propios parámetros juegan un papel importante en dicha convergencia. Una vez superada esta barrera, el problema de la demanda computacional hace que la convergencia a *policies* optimas requiera entrenamientos excesivamente largos. Sin embargo, *policies* cuasi-óptimas son alcanzables tras periodos de entrenamiento muy inferiores a los de RL, con las que, además, la capacidad de generalización es ampliamente superior en entornos similares al de aprendizaje. En entornos muy distintos la respuesta del agente es más limitada, como ya se concluía en (Banerjee et al., 2018) o (Dhiman et al., 2018) con algoritmos incluso más complejos como A3C. Además, queda clara de nuevo la incapacidad de explorar del agente con recompensas extrínsecas tradicionales (ver resultados entorno 3).

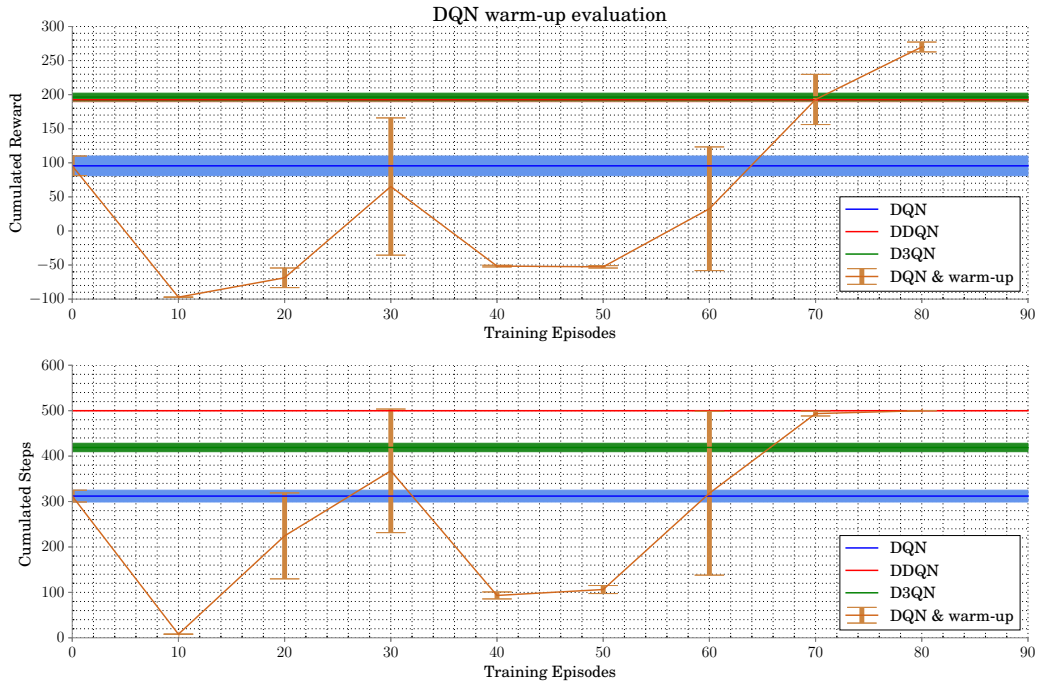


Figura 4.24: Recompensa (arriba) y pasos (abajo) de evaluación en el entorno 2 del agente DQN pre-entrenado durante distinto número de episodios (naranja), y valores obtenidos por los agentes DQN (azul), DDQN (rojo) y D3QN (verde) en el mismo escenario.

Capítulo 5

Conclusiones

> *"SL wants to work. Even if you screw something up you'll usually get something non-random back. RL must be forced to work. If you screw something up or don't tune something well enough you're exceedingly likely to get a policy that is even worse than random. And even if it's all well tuned you'll get a bad policy 30 % of the time, just because."*
>
> Andrej Karpathy.

5.1. Conclusiones

En este Trabajo Fin de Máster se han estudiado en profundidad los conceptos de SLAM activo, redes neuronales profundas y aprendizaje por refuerzo (profundo). Una amplia revisión del estado del arte de *Deep Reinforcement Learning* en exploración robótica se ha llevado a cabo en el capítulo 3, analizando numerosas soluciones adoptadas badas en tanto en *Deep Q-Networks* como en métodos más sofisticados: *Asynchronous Advantage Actor Critic*, redes recurrentes, etc. Se han evidenciado problemas en cuanto a la generalización a entornos distintos al de aprendizaje, y la confusión entre navegación pura y exploración. En este último ámbito, se ha propuesto una aproximación que contemple la inclusión de una métrica de la incertidumbre en la función de recompensa. Finalmente, se han implementado distintos algoritmos de *Q-learning* y *Deep Q-Networks* en un entorno de simulación complejo (Gazebo) empleando ROS, Tensorflow y Keras, entre otras herramientas. Los algoritmos de *Reinforcement Learning* han demostrado una limitada capacidad de generalización, así como la necesidad de largos entrenamientos. La inclusión en la función de recompensa de la entropía como métrica de la incertidumbre del mapa generado por un algoritmo de SLAM ha demostrado favorecer el aprendizaje de movimientos que disminuyan esta incertidumbre. Por otra parte, los algoritmos de *Deep Reinforcement Learning* han probado ser extremadamente sensibles a la configuración de sus numerosos parámetros, viéndose afectadas

la estabilidad y velocidad de aprendizaje, la demanda computacional o la convergencia, entre otros. Varios agentes con arquitecturas y configuraciones diferentes se han entrenado en un tiempo razonable, siendo capaces de extrapolar sus *policies* subóptimas a nuevos mapas de una forma satisfactoria siempre y cuando éstos sean "similares" al de aprendizaje. En esta aproximación *deep* se han intentado analizar qué ocurre exactamente en la red, cómo influye la variación de sus parámetros y otros comportamientos interesantes, e.g. el efecto que supondría tener un leve conocimiento *a priori* del nuevo entorno.

5.2. Trabajo Futuro

- Implementación y evaluación de algoritmos *policy-gradient*, e.g. A3C (*Asynchronous Advantage Actor-Critic*) o DDPG (*Deep Deterministic Policy Gradient*).
- Evaluación del entorno empleando imágenes como entrada (CNN) y añadiendo a éste objetos dinámicos.
- Evaluación de las recompensas con entropía en DRL.
- Uso de un software de SLAM más complejo.

Lista de Figuras

Lista de Tablas

Glosario

A3C	*Asynchronous Advantage Actor-Critic*.
ADRQN	*Action-based Deep Recurrent Q-Network*.
ANN	*Artificial Neural Network*.
BoW	*Bag of Words*.
BP	*Backpropagation*.
CNN	*Convolutional Neural Network*.
CPU	*Central Processing Unit*.
DBN	*Deep Belief Network*.
DBQN	*Deep Belief Q-Network*.
DDPG	*Deep Deterministic Policy Gradient*.
DDQN	*Double Deep Q-Network*.
DDRQN	*Deep Distributed Recurrent Q-Network*.
DL	*Deep Learning*.
DNC	*Differentiable Neural Computer*.
DNN	*Deep Neural Network*.
DQN	*Deep Q-Network*.
DRL	*Deep Reinforcement Learning*.
DRQN	*Deep Recurrent Q-Network*.

EKF	*Extended Kalman Filter*.
ELU	*Exponential Linear Unit*.
ER	*Experience Replay*.
FCL	*Fully Connected Layer*.
GD	*Gradient Descent*.
GPU	*Graphics Processing Unit*.
LSTM	*Long Short Term Memory*.
MDP	*Markov Decision Process*.
MLP	*Multi-Layer Perceptron*.
NTM	*Neural Turning Machine*.
PER	*Prioritized Experience Replay*.
PG	*Policy Gradient*.
POMDP	*Partially Observable Markov Decision Process*.
RBM	*Restricted Boltzman Machine*.
RBPF	*Rao-Blackwellized Particle Filter*.
ReLU	*Rectifier Linear Unit*.
RL	*Reinforcement Learning*.
RMSE	*Root Mean Squared Error*.
RNA	*Red Neuronal Artificial*.
RNN	*Recursive Neural Network*.
ROS	*Robotics Operating System*.
SA	*Stacked Autoencoders*.

SARSA	*State–Action–Reward–State–Action*.
SDA	*Stacked Denoising Autoencoders*.
SE	*State Estimator*.
SGA	*Stochastic Gracient Ascent*.
SGD	*Stochastic Gracient Descent*.
SL	*Supervised Learning*.
SLAM	*Simultanous Localisation and Mapping*.
SPLAM	*Simultanous Planning, Localisation and Mapping*.
TD	*Temporal Difference*.
TPU	*Tensor Processing Unit*.
VIO	*Visual Intertial Odometry*.
VO	*Visual Odometry*.

Referencias

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey of deep reinforcement learning. *ArXiv e-prints: 1708.05866*.

Bai, S., Chen, F., & Englot, B. (2017). Toward autonomous mapping and exploration for mobile robots through deep supervised learning. En *2017 ieee/rsj international conference on intelligent robots and systems (iros)* (pp. 2379-2384). IEEE.

Banerjee, S., Dhiman, V., Griffin, B., & Corso, J. J. (2018). Do deep reinforcement learning algorithms really learn to navigate?

Braziunas, D. (2003). Pomdp solution methods. *University of Toronto, Canada, Tech. Report*.

Büchler, D., Calandra, R., Schölkopf, B., & Peters, J. (2018). Control of musculoskeletal systems using learned dynamics models. (Vol. 3, *4*, pp. 3161-3168). IEEE.

Burusa, A. K. (2017). *Visual-inertial odometry for autonomous ground vehicles* (Tesis de maestría, KTH Royal Institute of Technology, Estocolmo, Suecia).

Caley, J. A., Lawrance, N. R., & Hollinger, G. A. (2016). Deep learning of structured environments for robot search. En *2016 ieee/rsj international conference onintelligent robots and systems (iros)* (pp. 3987-3992). IEEE.

Carrillo-Lindado, H. D. (2014). *Active slam: utility functions and applications* (Tesis doctoral, Universidad de Zaragoza, Departamento de Informática e Ingeniería de Sistemas).

Chen, F., Bai, S., Shan, T., & Englot, B. (2019). Self-learning exploration and mapping for mobile robots via deep reinforcement learning. En *Aiaa scitech 2019 forum* (p. 0396).

Chen, S. (2018). *Comparing drl methods for engineering applications* (Tesis de maestría, Otto-von-Guericke-Universität Magdeburg).

Chua, K., McAllister, R., Calandra, R., & Levine, S. (2018). Unsupervised exploration with deep model-based reinforcement learning.

Ciarfuglia, T. A., Costante, G., Valigi, P., & Ricci, E. (2014). Evaluation of non-geometric methods for visual odometry. (Vol. 62, *12*, pp. 1717-1730). Elsevier.

Concha, A., Loianno, G., Kumar, V., & Civera, J. (2016). Visual-inertial direct slam. En *2016 ieee international conference on robotics and automation (icra)* (pp. 1331-1338). IEEE.

Costante, G., Mancini, M., Valigi, P., & Ciarfuglia, T. A. (2016). Exploring representation learning with cnns for frame-to-frame ego-motion estimation. *IEEE Robotics and Automation letters*, *1*(1), 18-25.

Depeweg, S., Hernández-Lobato, J. M., Doshi-Velez, F., & Udluft, S. (2017). Decomposition of uncertainty in bayesian deep learning for efficient and risk-sensitive learning. *ArXiv e-prints: 1710.07283*.

Dhiman, V., Banerjee, S., Griffin, B., Siskind, J. M., & Corso, J. J. (2018). A critical investigation of deep reinforcement learning for navigation. *ArXiv e-prints: 1802.02274*.

Dosovitskiy, A., Fischer, P., Ilg, E., Hausser, P., Hazirbas, C., Golkov, V., ... Brox, T. (2015). Flownet: learning optical flow with convolutional networks. En *Proceedings of the ieee international conference on computer vision* (pp. 2758-2766).

Eade, E. & Drummond, T. (2006). Scalable monocular slam. En *2006 ieee computer society conference on computer vision and pattern recognition* (Vol. 1, pp. 469-476). IEEE.

Egorov, M. (2015). Deep reinforcement learning with pomdps. Stanford University, Stanford, CA, USA, Tech. Report.

Engel, J., Schöps, T., & Cremers, D. (2014). Lsd-slam: large-scale direct monocular slam. En *European conference on computer vision (eccv)* (pp. 834-849). Springer.

Farebrother, J., Machado, M. C., & Bowling, M. (2018). Generalization and regularization in dqn. *Computing Research Repository (CoRR)*, *abs/1810.00123*.

Foerster, J. N., Assael, Y. M., de Freitas, N., & Whiteson, S. (2016). Learning to communicate to solve riddles with deep distributed recurrent q-networks. *ArXiv e-prints: 1602.02672*.

Forster, C., Carlone, L., Dellaert, F., & Scaramuzza, D. (2017). On-manifold preintegration for real-time visual-inertial odometry. *IEEE Transactions on Robotics*, *33*(1), 1-21.

Forster, C., Pizzoli, M., & Scaramuzza, D. (2014). Svo: fast semi-direct monocular visual odometry. En *2014 ieee international conference on robotics and automation (icra)* (pp. 15-22). IEEE.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., ... Pietquin, O., et al., (2017). Noisy networks for exploration. *ArXiv e-prints: 1706.10295*.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., Pineau, J., et al., (2018). An introduction to deep reinforcement learning. *Foundations and Trends in Machine Learning*, *11*(3-4), 219-354.

Fridman, L. (2019). Mit deep learning. Recuperado el 11 de marzo de 2019, desde https://deeplearning.mit.edu/

Gao, X. & Zhang, T. (2017). Unsupervised learning to detect loops using deep neural networks for visual slam system. *Autonomous Robots*, *41*(1), 1-18.

Geiger, A., Ziegler, J., & Stiller, C. (2011). Stereoscan: dense 3d reconstruction in real-time. En *Intelligent vehicles symposium (iv), 2011 ieee* (pp. 963-968). IEEE.

Gibson, J. J. (1979). *The ecological approach to visual perception*. Houghton Mifflin.

Godard, C., Mac Aodha, O., & Brostow, G. J. (2017). Unsupervised monocular depth estimation with left-right consistency. En *Proceedings of the ieee conference on computer vision and pattern recognition (cvpr)* (Vol. 2, *6*, p. 7).

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *ArXiv e-prints: 1410.5401*.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., ... Agapiou, J., et al., (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, *538*(7626), 471.

Hausknecht, M. & Stone, P. (s.f.). Deep recurrent q-learning for partially observable mdps. En *2015 aaai fall symposium series*.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. En *Proceedings of the ieee conference on computer vision and pattern recognition (cvpr)* (pp. 770-778).

Hebb, D. (1949). *The organization of behavior*. Wiley.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, *79*(8), 2554-2558.

Ilg, E., Mayer, N., Saikia, T., Keuper, M., Dosovitskiy, A., & Brox, T. (2017). Flownet 2.0: evolution of optical flow estimation with deep networks. En *Proceedings of the ieee conference on computer vision and pattern recognition (cvpr)* (Vol. 2, p. 6).

Janisch, J. (2016). Deep learning for complete beginners: convolutional neural networks with keras. Recuperado el 8 de abril de 2019, desde https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, *101*(1-2), 99-134.

Karkus, P., Hsu, D., & Lee, W. S. (2017). Qmdp-net: deep learning for planning under partial observability. En *Advances in neural information processing systems* (pp. 4694-4704).

Kiefer, J. (1974). General equivalence theory for optimum designs (approximate theory). *The annals of Statistics*, 849-879.

Kim, D. K. & Chen, T. (2015). Deep neural network for real-time autonomous indoor navigation. *ArXiv e-prints: 1511.04668*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. En *Advances in neural information processing systems* (pp. 1097-1105).

Kümmerle, R., Grisetti, G., Strasdat, H., Konolige, K., & Burgard, W. (2011). G2o: a general framework for graph optimization. En *2011 ieee international conference on robotics and automation (icra)* (pp. 3607-3613). IEEE.

Lei, T., Giuseppe, P., & Ming, L. (2017). Virtual-to-real deep reinforcement learning: continuous control of mobile robots for mapless navigation. En *2017 ieee/rsj international conference on intelligent robots and systems (iros)* (pp. 31-36). IEEE.

Lei, T. & Ming, L. (2016). A robot exploration strategy based on q-learning network. En *2016 ieee international conference on real-time computing and robotics (rcar)* (pp. 57-62).

Leutenegger, S., Lynen, S., Bosse, M., Siegwart, R., & Furgale, P. (2015). Keyframe-based visual–inertial odometry using nonlinear optimization. *The International Journal of Robotics Research*, *34*(3), 314-334.

Li, R., Wang, S., Long, Z., & Gu, D. (2018). Undeepvo: monocular visual odometry through unsupervised deep learning. En *2018 ieee international conference on robotics and automation (icra)* (pp. 7286-7291). IEEE.

Linh Thai, H. (2018). *Deep reinforcement learning for pomdps* (Tesis de maestría, Technische Universitat Darmstadt, Alemania).

Liu, R. & Zou, J. (2018). The effects of memory replay in reinforcement learning. En *2018 56th annual allerton conference on communication, control, and computing (allerton)* (pp. 478-485). IEEE.

McCulloch, W. S. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115-133.

Meyer, D. (2018). Likelihood ratio policy gradients for reinforcement learning.

Minsky, M. L. & Papert, S. A. (1969). Perceptrons. MIT press.

Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., ... Hadsell, R. (2016). Learning to navigate in complex environments. *ArXiv e-prints: 1611.03673*.

Mitchell, T. M. (1997). *Machine learning*. New York, NY, USA: McGraw-Hill.

Mnih, V. [Volodymyr], Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. En *International conference on machine learning* (pp. 1928-1937).

Mnih, V. [Volodymyr], Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Ostrovski, G., et al., (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529.

Mohaimenian Pour, S., Martín, A., & Rivas-Montero, F. M. (2017). Deep reinforcement learning in robotics. Recuperado el 23 de mayo de 2019, desde https://jderobot.org/Deep_Reinforcement_Learning_in_Robotics

Muller, P. & Savakis, A. (2017). Flowdometry: an optical flow and deep learning based approach to visual odometry. En *2017 ieee winter conference on applications of computer vision (wacv)* (pp. 624-631). IEEE.

Mur-Artal, R., Montiel, J. M. M., & Tardós, J. D. (2015). Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, *31*(5), 1147-1163.

Mur-Artal, R. & Tardós, J. D. (2017a). Orb-slam2: an open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, *33*(5), 1255-1262.

Mur-Artal, R. & Tardós, J. D. (2017b). Visual-inertial monocular slam with map reuse. *IEEE Robotics and Automation Letters*, *2*(2), 796-803.

Nakajima, Y., Tateno, K., Tombari, F., & Saito, H. (2018). Fast and accurate semantic mapping through geometric-based incremental segmentation. En *2018 ieee/rsj international conference on intelligent robots and systems (iros)* (pp. 385-392). IEEE.

Nistér, D., Naroditsky, O., & Bergen, J. (2006). Visual odometry for ground vehicle applications. *Journal of Field Robotics*, *23*(1), 3-20.

Oh, J., Chockalingam, V., Singh, S., & Lee, H. (2016). Control of memory, active perception, and action in minecraft. *ArXiv e-prints: 1605.09128*.

Parisotto, E., Singh Chaplot, D., Zhang, J., & Salakhutdinov, R. (2018). Global pose estimation with an attention-based recurrent network. En *Proceedings of the ieee conference on computer vision and pattern recognition (cvpr) workshops* (pp. 237-246).

Peng, M., Wang, C., Chen, T., Liu, G., & Fu, X. (2017). Dual temporal scale convolutional neural network for micro-expression recognition. *Frontiers in Psychology*, *8*, 1745.

Piniés-Rodríguez, P. (2009). *Slam in large environments with wearable sensors* (Tesis doctoral, Universidad de ZaragozaA critical investigation of deep reinforcement learning for navigation).

Rodríguez-Arévalo, M. L. (2018). *On the uncertainty in active slam: representation, propagation and monotonicity* (Tesis doctoral, Universidad de Zaragoza).

Rodríguez-Arévalo, M. L., Neira, J., & Castellanos, J. Á. (2018). On the importance of uncertainty representation in active slam. *IEEE Transactions on Robotics*, *34*(3), 829-834.

Rojas, R. (1996). *Neural networks: a systematic introduction*. Springer-Verlag.

Rosenblatt, F. (1961). *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Parallel distributed processing: explorations in the microstructure of cognition, vol. 1. (Cap. Learning internal representations by error propagation, pp. 318-362). Cambridge, MA, USA: MIT Press.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. *ArXiv e-prints: 1511.05952*.

Shabbir, J. & Anwer, T. (2018). A survey of deep learning techniques for mobile robot applications. *ArXiv e-prints: 1803.07608*.

Shalev-Shwartz, S., Shammah, S., & Shashua, A. (2016). Safe, multi-agent, reinforcement learning for autonomous driving. *ArXiv e-prints: 1610.03295*.

SkyMind. (2018). A beginner's guide to deep reinforcement learning. Recuperado el 6 de marzo de 2019, desde https://skymind.ai/wiki/deep-reinforcement-learning

Smith, S. L., Kindermans, P.-J., Ying, C., & Le, Q. V. (2017). Don't decay the learning rate, increase the batch size. *ArXiv e-prints: 1711.00489*.

Smolyanskiy, N., Kamenev, A., Smith, J., & Birchfield, S. (2017). Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness. En *2017 ieee/rsj international conference on intelligent robots and systems (iros)* (pp. 4241-4247). IEEE.

Spark, C. (2017). Deep learning for complete beginners: convolutional neural networks with keras. Recuperado el 27 de junio de 2018, desde https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html

Sünderhauf, N., Brock, O., Scheirer, W., Hadsell, R., Fox, D., Leitner, J., ... Milford, M., et al., (2018). The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research*, *37*(4-5), 405-420.

Sünderhauf, N., Leitner, J., Upcroft, B., & Roy, N. (2018). Special issue on deep learning in robotics. *The International Journal of Robotics Research*, *37*(4-5), 403-404.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*(1), 9-44.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... Rabinovich, A. (2015). Going deeper with convolutions. En *Proceedings of the ieee conference on computer vision and pattern recognition (cvpr)* (pp. 1-9).

Tateno, K., Tombari, F., Laina, I., & Navab, N. (2017). Cnn-slam: real-time dense monocular slam with learned depth prediction. En *Proceedings of the ieee conference on computer vision and pattern recognition (cvpr)* (Vol. 2, pp. 6243-6252).

Thrun, S., Burgard, W., & Fox, D. (2005). *Probabilistic robotics*. The MIT Press.

Usenko, V., Engel, J., Stückler, J., & Cremers, D. (2016). Direct visual-inertial odometry with stereo cameras. En *2016 ieee international conference on robotics and automation (icra)* (pp. 1885-1892). IEEE.

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. En *Aaai conference on artificial intelligence*.

Vijayanarasimhan, S., Ricco, S., Schmid, C., Sukthankar, R., & Fragkiadaki, K. (2017). Sfm-net: learning of structure and motion from video. *ArXiv e-prints: 1704.07804*.

Viñal-Pons, J. (2012). *Localización y generación de mapas del entorno (slam) de un robot por medio de una kinect* (Tesis de maestría, Escola Tècnica Superior d'Enginyeria Informàtica Universitat Politècnica de València).

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. *ArXiv e-prints: 1511.06581*.

Watkins, C. J. & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3-4), 279-292.

Wulfmeier, M. (2018). On machine learning and structure for mobile robots. *ArXiv e-prints: 1806.06003*.

Zamora, I., Lopez, N. G., Vilches, V. M., & Cordero, A. H. (2016). Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo. *ArXiv e-prints: 1608.05742*.

Zhang, J., Tai, L., Boedecker, J., Burgard, W., & Liu, M. (2017). Neural slam: learning to explore with external memory. *ArXiv e-prints: 1706.09520*.

Zhao, C., Sun, L., Purkait, P., Duckett, T., & Stolkin, R. (2018). Learning monocular visual odometry with dense 3d mapping from dense 3d flow. En *2018 ieee/rsj international conference on intelligent robots and systems (iros)* (pp. 6864-6871). IEEE.

Zhelo, O., Zhang, J., Tai, L., Liu, M., & Burgard, W. (2018). Curiosity-driven exploration for mapless navigation with deep reinforcement learning. *ArXiv e-prints:1804.00456*.

Zhou, T., Brown, M., Snavely, N., & Lowe, D. G. (2017). Unsupervised learning of depth and ego-motion from video. En *Proceedings of the ieee conference on computer vision and pattern recognition (cvpr)* (Vol. 2, *6*, p. 7).

Zhu, P., Li, X., Poupart, P., & Miao, G. (2018). On improving deep reinforcement learning for pomdps. *ArXiv e-prints:1804.06309*.