Trabajo Control Inteligente de Sistemas Dinámico

Jon Armendariz, Anne Elorza, Jon Kerexeta, Asier López, Mikel Sánchez, Unai Saralegui, Oscar Serradilla, Mikel de Velasco, Andutz Zulaika

KISA 2016-2017

1 Introducción

En las próximas páginas se detalla un proyecto grupal llevado a cabo por estudiantes del Máster KISA en el contexto de la asignatura *Control Inteligente de Sistemas Dinámicos*. Más concretamente, se trata de un proyecto introductorio al mundo del aprendizaje automático aplicado al área de la robótica. Dado que las estrategias y métodos de aprendizaje automático conllevan muchas iteraciones para conseguir resultados aceptables, y por simpleza y versatilidad, se ha elegido llevar a cabo todos los experimentos en entorno simulado. Específicamente, en el resto del documento se relata nuestra lucha particular para conseguir que un coche recorra sin colisionar de la manera más veloz posible un circuito delimitado por paredes.

Comenzaremos el informe de la misma forma que comenzamos el trabajo, realizando un análisis de los simuladores disponibles para desarrollar el proyecto en la sección 2. Después plantearemos más detalladamente el problema, en el entorno de simulación elegido en la sección 3. En la sección 4 analizaremos la primera estrategia utilizada para intentar llegar a nuestro objetivo, la cual se basa en técnicas de aprendizaje de refuerzo aplicadas a sistemas que funcionan con redes neuronales. Tras no ser capaces de lograr nuestras aspiraciones con esta técnica, hemos probado otro enfoque al problema, en el cual profundizaremos en la sección 5. Finalizaremos este informe con los resultados obtenidos mediante ambas aproximaciones al problema al igual que una discusión de los mismos, en la sección 6.

2 Análisis de simuladores

Los simuladores de robots se usan para el diseño y testeo de algoritmos de control en diferentes plataformas. Se han seleccionado y comparado dos de los mejores simuladores de hoy en día. Estos simuladores son Gazebo y V-REP. Los simuladores de robots trabajan como puente entre la teoría de los robots y la robótica. Proporcionan un framework de carácter cognitivo que puede aplicarse a robots, realizando pequeños ajustes. Un simulador bien diseñado puede testear algoritmos, diseñar robots, realizar pruebas de regresión, y entrenar sistemas de AI (*Artificial Intelligence*) usando escenarios realistas.

2.1 Gazebo vs. V-REP

Los dos simuladores seleccionados para la comparación son Gazebo (solución Open-Source mantenida por Open Source Robotic Foundation) y V-REP (Solución comercial con una versión educativa gratuita proporcionada por Coppelia Robotics). Gazebo es un programa open source distribuido bajo la licencia Apache 2.0, tiene una interfaz muy amigable, diseñado para probar de forma rápida algoritmos y diseños de robots, y tiene una comunidad que crece a diario. Según estudios realizados, Gazebo es el simulador más conocido, mientras que V-REP es el que mejor y más cantidad de opciones ofrece (en cuanto a robots, escenarios, controladores,...),

presentando muchas más facilidades en lo que respecta a usabilidad e instalación. Ambos permiten trabajar con Python, Octave, Matlab, C++, Java, etc. Tanto V-REP como Gazebo cuentan con APIs de programación bastante completas que hacen más fácil la generación de programas.

2.1.1 Morfología del robot

Para la parte de renderizado, **Gazebo** cuenta con la ayuda de OGRE, que ofrece un entorno sencillo para el diseño orientado a objetos e independiente de la implementación 3D. Gazebo dispone de un editor que permite añadir figuras básicas (cilindros, esferas, cubos,...) u otras más elaboradas basadas en gráficos SVG o mayas 3D. Para unir figuras y conformar el robot, hay uniones en las que se puede definir el tipo de movimiento (Rotación, prismático, esférico,...). Además permite definir límites de movimiento, fuerza soportada, velocidad soportada, etc. Permite desarrollar prototipos propios además de modelos propios. **V-REP** dispone de un simulador 3D propio y es capaz tanto de editar los modelos como de importar y exportar modelos básicos.

2.1.2 Dinámica y cinemática

En estos simuladores también es posible la parametrización de los diferentes elementos que intervienen en el robot. Estos simuladores disponen de un conjunto amplio de clases matemáticas que permiten definir y operar, por ejemplo, con matrices, quaternios, planos, etc. En cuanto a la dinámica **Gazebo** es compatible con varios motores físicos entre los que se encuentran ODE, Simbody, Dart y Bullet. De igual modo **V-REP** cuenta con otros cuatro sistemas de dinámicas: Buller, ODE, Newton y Vortex.

2.1.3 Entorno

Gazebo dispone de una interfaz en la que se permite diseñar todo el entorno en el que operará el robot, dotando a todos los objetos las características de cuerpos rígidos con colisión. Además de colisiones propias de los objetos, se dispone de la posibilidad de añadir sensores de tipo láser, cámaras 2D o 3D, sensores de fuerza, de contacto, etc. Todos estos sensores pueden ser incluidos con ruido, consiguiendo así una simulación más real. Otro punto a favor es la posibilidad de realizar simulaciones remotas y en la nube.

En contraste, **V-REP** dispone de una interfaz integrada que nos permite diseñar todo el entorno y añadir nuestros modelos con la simple acción de coger y soltar. En cuanto al entorno, el simulador dispone de:

- Sistema de colisión,
- Sensores de proximidad, visión, etc., y
- Sistema de partículas y fluidos.

2.2 V-REP (Virtual Robot Experimentation Platform)

El simulador de robótica V-REP dispone de una interfaz o entorno de desarrollo (IDE) en la que cada modelo u objeto está basado en una arquitectura de control distribuido (puede ser controlado por un script propio, un plugin, una aplicación cliente remota a través de su API, un nodo ROS (Robot Operative System) etc). Esto hace ideal a la plataforma de simulación en un entorno multi-robot.

V-REP se usa para el desarrollo rápido de algoritmos, simulación de automatización en fábricas, para prototipado rápido y verificación, enseñanza de la robótica, etc. V-REP es multiplataforma, y permite la creación de contenido portable, escalable y de fácil mantenimiento: un simple fichero portable puede contener un modelo funcional completo (o escena), incluyendo código de control.

Tanto el simulador como las simulaciones son altamente personalizables, y para ello hay 6 aproximaciones de programación mutuamente compatibles y que pueden trabajar codo con codo. Cuando se habla de las 6 aproximaciones, se hace referencia a que la entidad de control, la escena, o el simulador en cuestión pueden situarse dentro de un script, un add-on, un plugin, una API remota, un nodo ROS o dentro de la arquitectura cliente-servidor.

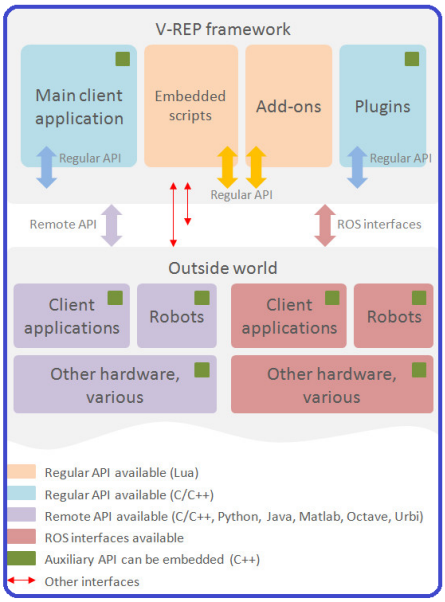


Figura 1: *Framework de V-REP.*

El módulo dinámico de V-REP integra cuatro motores físicos: the Bullet physics library, the Open Dynamics Engine, the Vortex Dynamics engine y the Newton Dynamics engine. Estos motores sirven para realizar de forma rápida y personalizable los cálculos dinámicos, simular el entorno real y la interacción entre objetos.

Por último, y de forma general, comentar que hay muchos escenarios, robots y funciones disponibles para este simulador. Hay desarrollados paquetes con distintas funcionalidades: para la detección de colisiones, cálculo de distancias mínimas, simulación de sensores de proximidad, simuladores de visión, planificación del movimiento, etc.

3 Manta y el problema

Habiendo buscado diferentes tipos de simuladores y después de haber analizado los pros y contras de cada implementación se decidió utilizar V-REP ya que consideramos que era la que mayores oportunidades ofrecía a la vez que nos habilitaba programar en lenguaje Python, lo cual buscamos desde el principio debido a la gran familiaridad de algunos de los miembros del equipo con dicho lenguaje.

Una vez escogidos el entorno de desarrollo y el lenguaje en el que se implementarán las funcionalidades necesarias así como los algoritmos para el aprendizaje, era el turno de definir un problema sobre el que trabajar y la forma de hacerlo. Después de valorar el tiempo con el que contaba cada miembro del equipo, decidimos acotar el problema en dificultad para que la tarea, independientemente del tiempo necesario para realizarlo, no supusiese una carga incompatible con las asignaturas en curso. Por lo tanto, después de varias ideas decidimos buscar un vehículo de cuatro ruedas con la finalidad de que éste aprendiera a moverse por un circuito de la forma más rápida posible.

Tal y como se ha mencionado, el problema en cuestión consiste en tratar un vehículo como si de un coche de carreras auto-dirigido se tratara. La idea inicial consistía en crear más de un circuito, entrenar el vehículo en alguno de ellos y ver cómo se comportaba al cambiar de circuito. Para crear el recorrido se han utilizado bloques que son facilitados por V-REP, colocándolos en diferentes posiciones y diferentes ángulos para crear el circuito que muestra la figura 2.

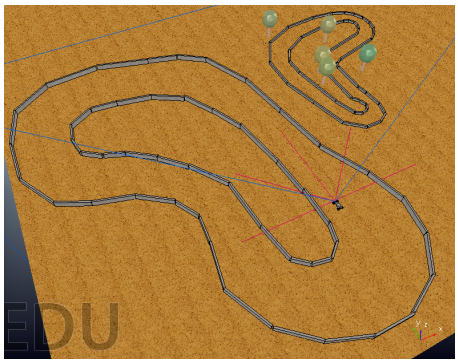


Figura 2: *Circuito diseñado en V-REP.*

Analizando la estructura del circuito diseñado (ver imagen 2) la sensórica a usar se basaba según nuestras ideas en sensores de proximidad ya implementados en V-REP y en visión por cámara, que, a partir de un momento dado, fue descartada por no obtener buenos resultados y requerir un cierto grado de cálculo. Por lo tanto, viendo que el vídeo no ayudaba y a la vez añadía complejidad a los modelos, ésta fue descartada; dejando a los sensores de proximidad como los ojos del robot. Estos sensores son su única visión del entorno, a parte de sensores de choque (definidos a partir de los sensores de proximidad) que fueron utilizados para parar las simulaciones cuando el vehículo quedaba atascado al generar las etiquetas.

Como ya se ha mencionado, buscamos un vehículo que encuentre la forma de conducir más óptima para recorrer un circuito. V-REP nos facilita modelos de más de un vehículo, tales como coches, tanques, aviones, etc. En nuestro caso hemos seleccionado el vehículo de cuatro ruedas *Manta*. Tal y como puede verse en la imagen 3, el vehículo cuenta con cuatro ruedas además de amortiguadores y una estructura; por lo demás, el vehículo no cuenta con ninguna funcionalidad extra a parte de las que nosotros añadamos.

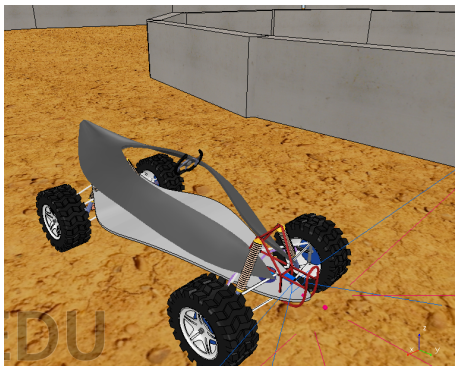


Figura 3: *Vehículo utilizado en nuestro modelo, Manta.*

3.1 Estructura del sistema

Explicadas las dos técnicas anteriores, hay que decir que nosotros hemos utilizado la técnica de *Policy Gradient Method* para el aprendizaje de nuestro sistema.

Primero empezaremos explicando el esquema funcional que nos encontramos para introducirnos en el problema. Siendo el objetivo crear un sistema que sea capaz de controlar un coche para completar vueltas a un

circuito. Dicho coche deberá tener ciertos sensores, los cuales serán las percepciones del sistema.

3.1.1 Sensores

Hay dos tipos de percepciones: aquellas que reflejan el estado del entorno (percepciones externas) y las que reflejan el estado del robot (percepciones internas); en nuestro caso del vehículo.

Como percepciones externas, hemos puesto varios sensores de proximidad en la parte delantera del vehículo con diferentes direcciones; son cinco sensores, dos que apuntan uno a cada lado en ángulo recto, otro mirando al frente y los otros dos en un punto intermedio entre los anteriores (a 45 grados). Los sensores de proximidad que tenemos ofrecen dos tipos de información: por una parte, si hay alguna interferencia en su rango de alcance (en caso de que la haya), la distancia a la que se encuentra el objeto más próximo en esa dirección y, por otra parte, la normal del plano del objeto que el sensor percibe. El alcance de todos los sensores es el mismo: 8 metros.

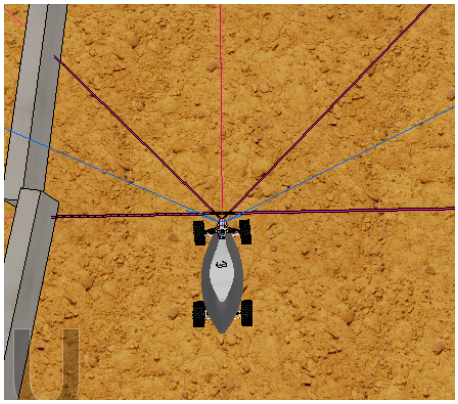


Figura 4: *Colocación de los sensores en el vehículo.*

En algunos experimentos, también dotamos al sistema de visión mediante una cámara. Aún así, esto generó información difusa y difícil de interpretar, y al no tener un criterio claro para discriminar esa información, en la mayoría de experimentos decidimos no utilizar la cámara como entrada sensorial.

En cuanto a las percepciones internas, *Manta* es capaz de procesar dos percepciones distintas: la velocidad a la que se mueve, y el ángulo de inclinación de sus ruedas delanteras (para percibir el sentido del giro). Es decir, esta situación es más o menos similar a la de una persona en un coche: mientras conducimos, lo mínimo necesario es información sobre la velocidad que llevamos y la posición del volante.

3.1.2 Actuadores

El sistema además de estar sensorizado, debe tener algún tipo de actuadores mediante los que pueda interactuar con el entorno. En este caso, como el objetivo es que el vehículo dé vueltas a un circuito, tenemos dos tipos de actuadores. El primero, actúa sobre el motor dándole más o menos potencia, y haciendo por tanto que el vehículo se mueva a mayor o menor velocidad. El segundo actuador, permite controlar el giro de las ruedas delanteras de manera que el vehículo pueda girar tanto hacia la izquierda como hacia la derecha, o bien mantenerse recto.

Estos dos actuadores se han discretizado para simplificar el control, de manera que cada actuador tiene siete estados posibles. Entre los siete diferentes modos de actuación del motor, el primero indica no ejercer fuerza sobre el motor (de forma que *Manta* decelera considerablemente debido a las distintas fricciones), y después cada estado siguiente indica mayor velocidad, la cual crece linealmente hasta el máximo. Por otro lado, en cuanto al ángulo de giro le damos la posibilidad de girar desde -40 grados hasta 40 grados. Pero, a lo largo de los 7 estados, no aplicamos un aumento lineal, sino que le hemos dado más sensibilidad cuando hay mucho giro

y cuanto menos giro queremos el salto es mayor. Los valores del giro de las ruedas en ángulos son: (-40, -24, -12, 0, 12, 24, 40).

3.1.3 Modelo para la toma de decisiones

Siendo ésta la estructura de nuestro sistema, ahora nos enfrentamos al gran problema de este proyecto, conseguir que *Manta* recorra el circuito sin colisionar con las paredes a la mayor velocidad posible. Se podría plantear una solución relativamente rápida y eficaz a este problema, utilizando un conjunto de reglas suficientemente grande para construir un sistema experto que diese vueltas sin demasiados problemas. De hecho, este enfoque muy probablemente daría mejores resultados que los experimentos hasta ahora llevados a cabo. Aún así, el hecho de intentar desarrollar un sistema que de alguna forma aprenda a llevar a cabo esta tarea resulta mucho más motivante, ya que emplearemos métodos más fácilmente generalizables que los sistemas expertos, y que además requieren de mucho menos conocimiento sobre el problema que vamos a tratar.

Ya sabemos qué percepciones tenemos y qué acciones podemos tomar; ahora hay que crear un sistema que mediante las percepciones obtenidas sea capaz de determinar cuál es la mejor acción que debe tomar para cumplir con sus objetivos. Para decidir qué maniobra ejecutar hemos utilizado las redes neuronales, concretamente las redes *feedforward*.

Dependiendo de las percepciones que deseemos procesar, hemos creado 3 distintas redes:

DNN1: Esta fue la primera red que ideamos y es la que percibe todos los sensores. Por una parte, toma como entrada la imagen que consta de 3 canales (*Red*, *Green*, *Blue*) y una matriz de 128 por 64 por canal de valores entre 0 y 1 ya que han sido previamente normalizados. Por otro lado, toma otros 27 valores distintos, también normalizados; las percepciones internas (el sentido de giro y la velocidad), y el resto de percepciones internas provenientes de los sensores de proximidad (un valor que indica si hay algo en el rango de alcance, la distancia a la que se encuentra y los tres valores que definen la normal del plano del punto que está al alcance).

Esta red procesa dos grupos de entradas de forma paralela que después se unen. Como primera entrada tenemos la imagen, que mediante dos secuencias de convoluciones y max-pooling reducimos su tamaño desde 128x64 hasta 6x2. A cambio, aumentamos el número de filtros: inicialmente contamos con tres (los canales RGB) y finalizamos con 7. La segunda entrada paralela agrupa el resto de entradas que simplemente se pasan por una capa totalmente conectada del mismo número de neuronas (27). Más tarde se juntan las dos redes para acabar en una capa de 64 neuronas y luego (de forma completamente conectada también) en la salida. La salida consta de 14 neuronas, una por acción posible (luego analizaremos como la interpretamos).

DNN2: Una vez probada la red anterior nos dimos cuenta de que tenía demasiados parámetros que optimizar, por ello optamos por quitar la visión y crear una red más simple. Esta red tomaba por entrada los 27 valores antes mencionados y mediante dos capas de 30 neuronas acababan en la salida.

DNN3: La red anterior seguía pareciendo tener demasiados parámetros a optimizar, por ello construimos una red bastante más simple que tomaba como entrada tan solo 7 valores: la percepción de velocidad, la de giro y las distancias al objeto más cercano que detectan los 5 sensores.

Estas tres redes comparten la salida, en la cual hay una neurona por acción posible a tomar. Las 14 son la salida de una función sigmoide, donde las primeras 7 se van a interpretar como una distribución de probabilidad sobre las siete acciones que puede tomar *Manta* para variar el giro de las ruedas y los últimos 7 serán las diferentes ejecuciones que pueden actuar sobre el motor. Estos dos conjuntos de distribuciones nos indican la probabilidad de tomar cada acción (siempre tomaremos una sobre el motor y otra sobre la dirección). Por tanto, debemos normalizar las dos distribuciones para que cumplan el requisito básico de cualquier distribución de probabilidad (que su suma sea 1). Esto se ha probado con dos conversiones diferentes:

- La primera normalización es una transformación lineal:

$$p_i = \frac{x_i}{\sum_{k=1}^{7} x_k} \tag{1}$$

  donde i es un índice que identifica cada una de las siete posibles acciones a tomar en cada uno de los dos actuadores (por tanto $1 \leq i \leq 7$), x_i representa la salida de la red correspondiente a la posible acción i, y p_i indica la probabilidad de tomar la acción i.

- La segunda normalización es una transformación exponencial, la cual es dependiente de un parámetro α. Cuanto mayor sea α, más brusca será la normalización. Es decir, cuanto mayor sea α, mayor será el valor normalizado del elemento con mayor valor de salida de la red en relación con el resto:

$$p_i = \frac{e^{\alpha x_i}}{\sum_{k=1}^{7} e^{\alpha x_k}} \tag{2}$$

Estas dos normalizaciones tomarán especial importancia en la fase de entrenamiento, ya que se tomarán decisiones de forma no determinista. Así dependiendo de la normalización (y del valor de α en el caso de la normalización exponencial) se conseguirá que el comportamiento de *Manta* sea más o menos predecible, más o menos determinista.

4 Entrenamiento mediante técnicas de Reinforcement Learning

Como en nuestro sistema la toma de decisiones se lleva a cabo con una red neuronal, el método de entrenamiento debe encargarse básicamente de conseguir ejemplos de entrenamiento para poder ajustar los pesos y bias de ésta. Es decir, debemos obtener un conjunto suficientemente amplio de parejas de percepciones y acciones deseables a tomar. En líneas generales, utilizaremos dos metodologías diferentes para obtener este conjunto de muestras de entrenamiento para la red. La primera, donde ha ido la inmensa mayoría de nuestro esfuerzo, es una interpretación de una técnica de aprendizaje por refuerzo conocida como *Policy Gradient Methods* (PGM)[1]. Hemos tenido una gran cantidad de inconvenientes que salvar al trabajar con esta técnica, así que finalmente hemos decidido probar también con un método muy distinto. Se trata de sustituir a la red neuronal por una persona durante el entrenamiento mediante PGM, para básicamente hacer que la red interiorice las reglas humanas. Podemos entender esto como una manera indirecta y más abstracta de generar un sistema experto. Analizaremos primero cómo hemos planteado el PGM, y después explicaremos con más detalle como hemos introducido el conocimiento humano en el sistema.

4.1 Implementación e interpretación de los *Policy Gradient Methods*

El sistema de aprendizaje desarrollado consta de varios módulos para lograr ejemplos de entrenamiento para la red. Todo comienza con una o (normalmente) varias simulaciones, con las que se consigue un historial de percepciones y acciones tomadas. Por supuesto, los *Policy Gradient Methods*, como buen sistema de aprendizaje por refuerzo, dependen de una función *reward* que evalúa cómo de apropiadas son las acciones que se toman. Por tanto, debemos definir esta función, la cual asignará el valor de recompensa a cada una de las parejas percepción-acción del historial de la simulación. Una vez tenemos el conjunto de las parejas percepción-acción valoradas, se procede a generar los ejemplos de entrenamiento para la red. Es aquí donde entran en juego los gradientes.

[1] Policy Gradient Methods: `http://karpathy.github.io/2016/05/31/rl/`

El objetivo es, para un trío percepción-acción-recompensa, crear un gradiente en la salida de la red que actualice (mediante el clásico algoritmo de propagación hacia atrás[2]) los pesos y bias de la red neuronal. El mencionado gradiente debería favorecer la acción ante la percepción si la recompensa ha sido alta, y desfavorecerla (o al menos no favorecerla) en el caso de que haya sido una mala decisión. Bajando varias capas en el problema y pasando a temas de implementación, esta técnica así planteada tiene el problema de que hay que programar el entrenamiento de la red manualmente (al menos parcialmente) para introducir de forma manual el gradiente. Teniendo en cuenta el gran número de librerías hoy en día existentes para el entrenamiento de redes neuronales con parejas entrada-salida, nuestra decisión ha sido no implementar directamente los PGM. En su lugar, generaremos los gradientes deseados de forma indirecta. Al final, cuando estamos entrenando una red neuronal en condiciones estándar, los parámetros de la red se actualizan de acuerdo con el gradiente y una función de error dada que indica como de bien se ajusta la salida de la red a la salida deseada. Por tanto podemos conseguir los gradientes que deseamos fijando una función de error para la red, y generando las salidas deseadas (de acuerdo a algún criterio) a partir de la acción tomada y la recompensa obtenida. Una vez generadas estas salidas deseadas, podemos entrenar la red fácilmente utilizando casi cualquier librería sobre redes neuronales, mediante los conocidos algoritmos de descenso de gradiente y propagación hacia atrás.

A continuación se describen con detalle los módulos desarrollados para completar este sistema de entrenamiento de la red encargada de tomar las decisiones de *Manta*.

4.2 Reward

Tal y como hemos explicado, dependiendo del *reward* penalizaremos o favoreceremos una acción tomada a partir de un estado (consideramos como estado los valores de los sensores en ese momento). Para definir los *rewards*, tenemos que valorar las acciones tomadas por *Manta* en cada instante. Como lo ideal es que le dé vueltas al circuito lo más rápido posible, lo primero que se nos ocurre es calcular el tiempo necesario para dar una vuelta. Como no disponemos de GPS pero sí de sensores de velocidad, nos aprovecharemos de estos últimos sensores y haremos algo equivalente: calcular la velocidad media que ha mantenido *Manta* en cierto intervalo de tiempo. De esta manera, si la velocidad media es alta, habrá recorrido mucha distancia en poco tiempo. Hemos tenido en cuenta que las paredes del circuito están lo suficientemente cerca para evitar que este dando vueltas sobre sí mismo. Además los choques son penalizados de forma explicita, si nos chocamos se reduce el *reward* a una cuarta parte.

Al calcular la velocidad media en un periodo de tiempo no estamos teniendo en cuenta solo el último estado y su correspondiente actuación, sino que tenemos en cuenta cada estado-actuación de cada instante de tiempo de dicho periodo. Esta lógica recae en que un buen *reward* no sólo depende de la última acción. Es posible que en cierto estado, por mucho que se escoja la mejor decisión, el resultado sea malo: si estamos conduciendo a 80km/h y a un metro de nosotros tenemos la pared, es inevitable el choque. Es momentos atrás donde teníamos que haber elegido una buena acción para evitar el choque.

Por esta razón, a un estado-acción, se le asignará el *reward* calculado después de evaluar la velocidad durante el estado de tiempo en cuestión y los instantes siguientes. De esta forma, si una acción ha sido la causante de un choque, se penalizara más tarde, y viceversa, si ha evitado un choque más adelante, mantendrá una velocidad media mayor que si hubiese tomado la mala decisión. Así, después de una simulación (recogida de datos), tendremos: estados, las acciones tomadas con estos estados, y las valoraciones de estas acciones.

4.3 Simulación y recolecta de datos

Con el objetivo de crear un vehículo capaz de tomar las acciones más óptimas de cara a recorrer un circuito de la forma más rápida posible, tenemos que generar un conjunto de ejemplos de entrenamiento para entrenar la

[2]Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015. `http://neuralnetworksanddeeplearning.com/`

red neuronal. Para ello se recogen datos acerca de las percepciones del robot con una frecuencia de 10Hz.

Para generar un conjunto de datos lo más general posible el *Manta* ha sido colocado en tres posiciones diferentes de manera aleatoria, posicionándose en sentidos aleatorios. Para generar el conjunto de entrenamiento se simula desde cada posición durante un máximo de un minuto siendo la simulación detenida al ocurrir un choque contra uno de los muros. Esto se realiza en tandas de veinte simulaciones con las que se generan los conjuntos de entrenamiento. Con objetivo de generalizar el modelo se ha diseñado una función que añade ruido a las recepciones sensoriales.

De esta manera, al entrenar los modelos, definiremos como entrada los estados, y como salidas, las acciones tomadas pero modificadas dependiendo del *reward*. Si repetimos la simulación después de entrenar, se conseguirán mejores datos. Y así sucesivamente, por lo que, *Manta* debería de ir aprendiendo progresivamente.

4.4 Generación automática de etiquetas

Una vez realizadas las simulaciones y haber recogido los datos, el siguiente paso es pensar cómo generar etiquetas para un conjunto de acciones. Tal y como se ha mencionado, a cada estado-acción se le asigna un *reward* dependiendo del efecto de esa acción en la evolución del movimiento futuro. La idea que hemos utilizado para implementar el etiquetado se basa intuitivamente en intentar alejarnos de una acción cuando el *reward* asociado sea pequeño y, en cambio, reforzar la acción realizada cuando sea grande. Con intención de llevar al papel esa idea y de poder modelarla, se propone una aproximación mediante funciones gaussianas.

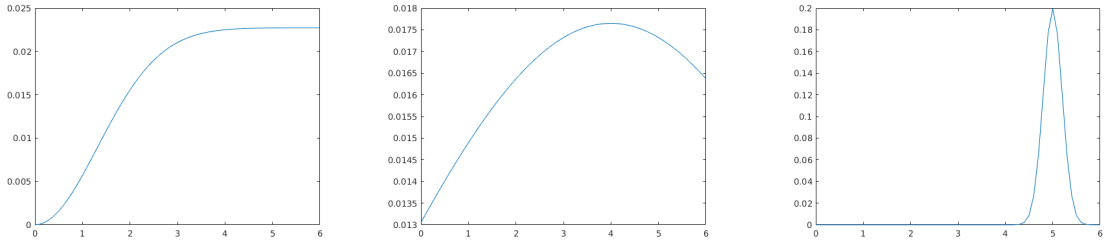
En general dada una acción y su *reward* asociado, el primer paso realizado ha sido relacionar la desviación estándar para una función gaussiana con el *reward* asociado a una acción concreta. Para ello se aplica una transformación lineal que haga que acciones con *reward* bajo conduzcan a desviaciones altas, para alejarnos de la acción tomada y al contrario asignando desviaciones bajas a aquellas acciones que tengan asociado un *reward* alto.

Después se centra la función gaussiana en el índice de acción que se ha tomado. En aquellos casos donde la acción tomada haya sido muy mala proponemos invertir la función gaussiana para que esa acción no sea considerada. Por último se normalizan las salidas para obtener las probabilidades.

Dada una acción y su *reward*, el vector de probabilidades de las posibles acciones a tomar se calcula de este modo:

- A partir del *reward* se calcula la desviación típica de la función gaussiana que se va a usar. Para ello, se aplica una transformación lineal, que a acciones con *reward* muy bajo les asigna desviaciones típicas muy altas, para alejarnos lo máximo posible de la acción. Por otro lado, a acciones muy buenas, les asigna desviaciones típicas bajas, tal y como se muestra en las figuras.
- Centramos la función gaussiana que va a definir las probabilidades en el índice de la acción que se ha tomado. En caso de que la acción haya sido mala, se invierte la campana de Gauss, para que la acción tomada no se considere.
- Normalizamos y discretizamos la salida para obtener valores que al sumar uno puedan representar probabilidades.

En las gráficas 5 pueden observarse diferentes funciones gaussianas definidas mediante la metodología anteriormente propuesta.



(a) Reward: 0.25.
Índice de la acción: 0.

(b) Reward: 2.
Índice de la acción: 4.

(c) Reward: 3.
Índice de la acción: 5.

Figura 5: Diferentes funciones gaussianas dependiendo del reward y el índice de acción

5 Modelado de la toma de decisiones de humanos en este problema

Siguiendo el desarrollo principal no hemos conseguido que el robot aprenda de forma automática a conducir por los circuitos. Por ello, proponemos un desarrollo alternativo en el que cambiamos el enfoque del problema, revisando las premisas tomadas como base para realizar el desarrollo principal.

Concretamente, revisamos la definición de inteligencia y más concretamente el significado de la inteligencia aplicada a robots. Razonamos que el ser humano entiende que los robots actúan de forma inteligente cuando se parecen a la forma en la que actuarían las personas para resolver un problema, adaptándose al entorno, explorando y aprendiendo, incluso mejorando los fallos que podemos cometer los humanos. Tomando este razonamiento como base, el nuevo enfoque consiste en enseñarle al robot la forma en la que una persona resolvería el problema, transmitiéndole las reglas implícitas que hemos aprendido los humanos, basadas en toda una vida de aprendizaje, experiencia y capacidades obtenidas mediante el desarrollo evolutivo de la especie humana. Por lo tanto, en este desarrollo hemos evitado crear un entorno artificial en el que el robot aprendería a resolver el problema interactuando de forma aleatoria hasta que casualmente tomase decisiones correctas y aprendiese iterativamente, confiando en la aleatoriedad y en la imprecisión del sistema de aprendizaje artificial.

Aunque los seres humanos seamos imperfectos y los robots no puedan ser perfectos aprendiendo de nosotros, lo que si pueden hacer es beneficiarse de aprender nuestro comportamiento lo mejor que puedan y luego intentar mejorarlo por su cuenta, evitando de este modo el consumo innecesario de recursos y tiempo para aprender el problema desde cero. Este sistema de aprendizaje es generalizable a cualquier tarea que los humanos seamos capaces de llevar a cabo pero nos sea imposible o muy difícil de explicar mediante reglas.

Para realizar este periodo de aprendizaje alternativo, se ha generado una plataforma de simulación en la que usuarios humanos eligen entre las acciones que puede tomar el robot a entrenar para recorrer el circuito sin chocarse. Las acciones se toman en tiempo real, viendo en cada momento que repercusión tienen las acciones tomadas y teniendo la posibilidad de corregir los errores en la siguiente decisión tomada.

Los usuarios cuentan con una perspectiva aérea del circuito por lo que tienen más información que el robot (como hemos especificado anteriormente, el robot sólo cuenta con sensores de proximidad), pero se espera que el robot sea capaz de tomar decisiones sólo con sus sensores, cosa que a los humanos nos resulta muy difícil. Que el robot aprenda a manejarse solo por el circuito es una tarea complicada y más si tiene unos sensores tan limitados, pero aprender de los humanos facilita enormemente la tarea.

Los usuarios siguen dos modos de conducir distintos para que el robot aprenda dos comportamientos distintos:

1. Conducción de forma que el robot recorre la mayor distancia posible en el tiempo de simulación. Este tipo de conducción sigue el objetivo general: recorrer el circuito de la forma más rápida posible.
2. Conducción de evitar obstáculos, acercándose a las paredes y corrigiendo la trayectoria. De esta forma, el

robot aprenderá a corregir su trayectoria para no chocarse en caso de que previamente haya tomado una decisión equivocada que le lleve a chocarse contra una pared. Se trata de un objetivo secundario pero es necesario para cumplir el objetivo general mencionado en el comportamiento anterior.

Durante la conducción de los usuarios en el entorno simulado se generan etiquetas de entrenamiento para la red neuronal que tomará las decisiones del robot una vez entrenada. Las etiquetas de entrenamiento contienen los datos de entrada de la red (generados por los sensores del robot cada instante de tiempo) y las decisiones que ha tomado el usuario que lo estaba guiando (actuadores del robot). Una vez que todos los usuarios han terminado la ejecución, se juntan los datos de entrenamiento y se entrena el robot.

6 Resultados, problemas, conjeturas y trabajo futuro

Como hemos ido mencionando en distintas ocasiones, la opción de aprendizaje de refuerzo mediante *Policy Gradient Methods* no ha resultado válida para conseguir que *Manta* aprenda a realizar circuitos satisfactoriamente. Nuestras hipótesis de este fracaso, entre comillas, son diversas. En primer lugar, puede que las entradas sensoriales de las que *Manta* ha sido dotado no sean suficientes para llevar a cabo la tarea a realizar, si bien es más probable que lo que *Manta* tenga sea un exceso de información. Por ejemplo, incorporar el RGB de todos los píxeles parece haber sido excesivo, ya que no contamos con ejemplos de entrenamiento suficientes como para poder entrenar una red que procese adecuadamente tantos valores, luego la cámara ha supuesto básicamente una fuente de ruido al sistema.

Muy relacionado con la cantidad de datos a procesar está la forma de procesarlos, es decir, la red neuronal utilizada. Anteriormente hemos hablado de las tres distintas estructuras utilizadas. La primera fue descartada rápidamente, ya que estaba preparada para procesar, además del resto de percepciones externas e internas, la ruidosa información que contenía la cámara. Entre las otras dos, no apreciamos demasiadas diferencias, ambas eran mejores que la primera, pero a la vez no se consiguió ningún resultado aceptable con ninguna.

El algoritmo de entrenamiento, por otra parte, también es mejorable. De hecho, todas las partes del algoritmo por separado tienen ciertos problemas, que después al unir todos los módulos se suman y acaban inutilizando la metodología. Nuestras conjeturas al respecto son las siguientes. En relación al *reward*, es probablemente suficiente, pero no es perfecto. Aplicamos el *reward* (la velocidad media) a una acción dada teniendo en cuenta varios instantes de tiempo anteriores y posteriores al instante correspondiente. Así, el *reward* depende, a parte de lo buena o mala que ha sido dicha acción, de la dificultad del tramo. Es decir, el *reward* será menor en un tramo de curvas que en un tramo más rápido si las acciones tomadas han sido igual de buenas.

Después, el método de generación de etiquetas puede que tampoco sea óptimo, a lo mejor el hecho de, si una acción ha resultado ser errónea enseñarle a la red a tomar opciones contrarias también sea una fuente de ruido. Quizás sería mejor simplemente entrenar la red con acciones que han resultado ser buenas, y cuanto mejores hayan sido, repetirlas (con ruido para prevenir algo el *overfitting*) para que la red incorpore esos conceptos más arraigadamente.

Para acabar con el repaso de nuestro algoritmo de aprendizaje por refuerzo, hemos de decir que intuimos que es necesario cierto mecanismo de control que evite degradar el rendimiento del sistema según avanzan las iteraciones. Tal y como esta planteado ahora mismo el algoritmo, si en una iteración dada se generaran ejemplos de entrenamiento inadecuados, sin duda se empeorará el comportamiento de *Manta* en la iteración posterior. Claramente este hecho ha tenido un impacto negativo en la fase de aprendizaje. Cuando hemos puesto a entrenar a *Manta*, en muchos casos ha aprendido a trazar alguna curva, pero después de iteraciones en las que (por el no determinismo de la toma de decisiones) su comportamiento no ha sido ideal, se le olvida cómo trazar la misma curva. Este *fenómeno* se puede detectar fácilmente haciendo algún tipo de media de los *rewards* obtenidos en cada iteración, y aplicando algún criterio para descartar algunas iteraciones perjudiciales.

El segundo método ha sido mucho más efectivo. Ha bastado con una persona *jugando* con *Manta* alrededor de un cuarto de hora para que éste comience a dar sus primeras vueltas sin chocarse. Viendo la efectividad de

este método, se nos ocurren distintos enfoques híbridos al problema. Por ejemplo, quizás inicializando el algoritmo de aprendizaje mediante los *Policy Gradient Methods* con una red preentrenada de esta forma conseguimos hacer funcionar el aprendizaje por refuerzo. En cualquier caso, antes habría que solucionar los problemas de estabilidad que éste tiene, dado que no es permisible que según avanzan las iteraciones empeore el comportamiento.

En cualquier caso, en líneas generales, este trabajo ha resultado ser una exigente introducción al mundo del aprendizaje automático y del *Deep Learning* aplicado a los robots. Nos hemos encontrado con muchos obstáculos que resolver, algunos los hemos superados, pero otros requerirían más trabajo y tiempo, y del último no disponemos en exceso. Alejándonos de las especificaciones del problema, este trabajo ha supuesto un reto que nos ha llevado a trabajar competencias como las de organización y trabajo en grupos relativamente grandes. Esto puede que nos haya ralentizado en algún momento debido a las distintas disponibilidades y habilidades de los integrantes, pero sin duda nos ha permitido conocer otras formas más complejas de trabajo, y también aprender cómo organizarnos y compenetrarnos. Así, estamos orgullosos de haber intentado que este proyecto funcionase lo mejor posible, y por supuesto agradecemos la oportunidad que se nos ha otorgado para realizarlo.