

MODELO DE ENTRENAMIENTO DE UN SISTEMA EXPERTO BASADO EN REGLAS MEDIANTE APRENDIZAJE POR REFUERZO APLICADO A LA GENERACIÓN DE TRAYECTORIAS

Milton Vicente Calderón Coy

Universidad Distrital Francisco José de Caldas
Facultad de Ingeniería
Maestría en Ciencias de la Información y las Comunicaciones
Bogotá D.C., Colombia
2021

MODELO DE ENTRENAMIENTO DE UN SISTEMA EXPERTO BASADO EN REGLAS MEDIANTE APRENDIZAJE POR REFUERZO APLICADO A LA GENERACIÓN DE TRAYECTORIAS

Milton Vicente Calderón Coy

Trabajo de grado presentado como requisito parcial para optar por el título de:
Magíster en Ciencias de la Información y las Comunicaciones

Director:
Ing. Esperanza Camargo Casallas PhD

Grupo de investigación:
DIGITI

Universidad Distrital Francisco José de Caldas
Facultad de Ingeniería
Maestría en Ciencias de la Información y las Comunicaciones
Bogotá D.C., Colombia
2021

A mi familia

Agradecimientos

En primer lugar a Dios, por todas sus bendiciones.

A mi madre por su cariño y amor incondicional, así como por su ejemplo de trabajo duro y horadez.

A mi hermano por su apoyo y sus aportes desde una óptica diferente.

A mi directora de tesis, Dra. Esperanza Camargo por su paciencia y guía en la realización de este trabajo.

A mis amigos Cesar, Diego y Noé por su amistad a lo largo de los años.

Resumen

En robótica móvil la tarea de generación de trayectorias hacía un punto de destino ha sido abordada desde diversos ángulos con diferentes algoritmos. Entre ellos pueden encontrarse búsquedas heurísticas, grafos hasta redes neuronales y algoritmos de enjambre. Para algunos de estos es necesaria la generación de un conjunto de nodos los cuales pueden estar organizados de múltiples formas a lo largo del entorno por el cual se desplazará el robot. Otro tipo de enfoque se centra en la naturaleza del entorno que puede ser total o parcialmente conocido, estático o dinámico, lo que hace que la ruta pueda ser planeada con antelación o deba ser trazada y ajustada al momento de realizar el recorrido hacia el punto de destino.

La inteligencia artificial es un campo relativamente nuevo que nació en los años cincuenta. En la actualidad una de sus ramas más interesantes es el aprendizaje por refuerzo. Esta rama consiste en un tipo de aprendizaje en el cual el sistema de inteligencia artificial interactúa con un ambiente a partir del cual se puede generar conocimiento de manera automática y paulatina.

En la actualidad existe un sin número de sistemas de inteligencia artificial que funcionan mediante complejas redes neuronales, en los cuales no es fácil determinar si el modelo detrás de sus decisiones presenta algún tipo de sesgo a la hora de analizar los datos. Por ejemplo, la clasificación del riesgo crediticio de una persona en función de variables tales como su raza o género, dada la poca interpretabilidad de dicho modelo. Por otro lado, existen otras metodologías que tienen una gran explicabilidad del modelo, como los árboles de decisión o sistemas expertos basados en reglas.

En la navegación estratosférica, resulta todo un reto orientar un vehículo en la fase de descenso teniendo en cuenta su velocidad de caida, así como el comportamiento aleatorio de las variables ambientales a las que se enfrenta.

Por esto, se desarrolló un sistema de entrenamiento de un sistema experto basado en reglas, el cual es entrenado mediante aprendizaje por refuerzo. Con ello en mente, fue necesario generar un contexto de entrenamiento que implemente la tarea que se pretende resolver mediante el sistema experto. Este entorno genera los desafíos que aprenderá el sistema experto. El sistema propuesto presentaría una ventaja sobre otros modelos: la posible intervención humana en el conjunto resultante de reglas, y, por ende, en el modelo que controla el sistema experto que toma las decisiones. Sin embargo, la posibilidad de realizar una auditoría al sistema y una posible modificación derivará en un cambio en la capacidad del sistema para resolver la tarea.

Como resultado se obtuvo un sistema de entrenamiento por refuerzo mediante el cual se obtiene un conjunto de reglas. Estas reglas contienen el conocimiento requerido por el sistema experto para establecer la trayectoria desde el punto de inicio hasta el punto de destino evitando salir del entorno, y esquivando los obstáculos en un 76.2 % de los episodios de validación ejecutados.

Palabras clave: Inteligencia artificial, Aprendizaje por refuerzo, Sistemas expertos, Generación de trayectorias.

Abstract

In mobile robotics, the task of generating trajectories to a destination point has been approached from various angles with different algorithms. These include heuristic search, graphs, neural networks, and swarm algorithms. Some of these require the generation of a set of nodes that can be organized in multiple ways throughout the environment through which the robot will move. Another type of approach focuses on the nature of the environment which may be fully or partially known, static or dynamic, which means that the route may be planned in advance or must be plotted and adjusted as it makes its way to the destination point.

Artificial intelligence is a relatively new field that was born in the 1950s. Currently, one of its most interesting branches is reinforcement learning. This branch consists of a type of learning in which the artificial intelligence system interacts with an environment from which knowledge can be generated automatically and gradually.

There are currently a large number of artificial intelligence systems that work by means of complex neural networks, in which it is not easy to determine whether the model behind their decisions presents some kind of bias when analyzing the data. For example, the classification of a person's credit risk based on variables such as race or gender, given the poor interpretability of such a model. On the other hand, there are other methodologies that have high model explainability, such as decision trees or rule-based expert systems.

In stratospheric navigation, it is a challenge to guide a vehicle in the descent phase taking into account its fall speed, as well as the random behavior of the environmental variables it faces.

Therefore, a rule-based expert system training system was developed, which is trained by reinforcement learning. With this in mind, it was necessary to generate a training context that implements the task to be solved by the expert system. This environment generates challenges to be learned by the expert system. The proposed system would present an advantage over other models: the possible human intervention in the resulting set of rules, and, therefore, in the model that controls the expert system that makes the decisions. However, the possibility of auditing the system and possible modification will result in a change in the system's ability to solve the task.

As a result, a reinforcement training system was obtained through which a set of

rules is obtained. These rules contain the knowledge required by the expert system to establish the trajectory from the starting point to the destination point, avoiding leaving the environment and avoiding obstacles in 76.2 % of the validation episodes executed.

CONTENIDO

RESUMEN III

ABSTRACT V

CONTENIDO VII

Lista de figuras XI

Lista de tablas XII

1. INTRODUCCIÓN 1

1.1. Planteamiento del problema 2
1.2. Formulación del problema 4
1.3. Objetivos 4
1.3.1. Objetivo general 4
1.3.2. Objetivos específicos 4

2. MARCO DE REFERENCIA 5

2.1. Inteligencia artificial 5
2.2. Redes neuronales 7
2.2.1. Topologías 9
Perceptrón 9
Perceptrón multicapa 9
2.2.2. Funciones de activación 10
Lineal(Linear) 10
Lineal rectificada(Rectified Linear Unit) 10
Sigmoide(Sigmoidal) 10
Tangente hiperbólica(hyperbolic tangent) 10
2.2.3. Aprendizaje 10
Supervisado 10
No supervisado 11

2.3. Aprendizaje por refuerzo 11
2.3.1. Elementos del aprendizaje por refuerzo 11
Entorno 11
Agente 12
Estado 12
Recompensa 13
2.3.2. Tipos de aprendizaje 13
Métodos basados en tablas 13
Métodos aproximados 14
2.4. Sistemas expertos 15
2.4.1. Elementos de un sistema experto 16
Motor de inferencia 16
Base de conocimiento 17
Reglas 17
Hechos 17
Mecanismo de aprendizaje 17
Interfaz de usuario 18
2.5. Generación de trayectorias 18
2.5.1. Técnicas de hoja de ruta 18
Grafos de visibilidad 19
Diagramas de Voronoi 19
2.5.2. Algoritmos de decomposición de celdas 20
2.5.3. Métodos de potenciales artificiales 21

3. DESARROLLO **22**
3.1. Componentes necesarios 22
3.1.1. Componentes entrenamiento por refuerzo 22
3.1.2. Componentes de entrenamiento por refuerzo de un sistema experto 24
3.2. Interacción entre los componentes 26
3.2.1. Interacción en entrenamiento por refierzo 26
Determinar acción 28
Almacenar datos 29
Realizar entrenamiento 29
3.2.2. Interacción propuesta en entrenamiento por refuerzo 30
Determinar acción 30
Almacenar datos 30
Realizar entrenamiento 32

3.3. Metodología propuesta 34
3.3.1. Generación de reglas 34
Algoritmo de generación de reglas 35

4. RESULTADOS **40**
4.1. Entorno de entrenamiento 40
4.2. Sistema de entrenamiento 45
4.3. Modelo de entrenamiento 48

5. CONCLUSIONES Y TRABAJOS FUTUROS **53**

BIBLIOGRAFÍA **55**

Anexos **59**

A. Código fuente **60**
A.1. Estructura del proyecto 60
A.2. Código fuente 61
A.2.1. lib 67
A.2.2. lib/agents 77
A.2.3. lib/agents/expertSystem 93
A.2.4. lib/agents/nn 109
A.2.5. lib/envs 116
A.2.6. lib/utils 131
A.2.7. test 133
A.2.8. tmp 133

B. Archivos resultantes **135**
B.1. Archivo de definición 135
B.2. Archivo XML 136

Lista de figuras

2.1. Neurona biológica 7
2.2. Diagrama de una neurona artificial 7
2.3. Perceptrón multicapa 9
2.4. Interacción agente-entorno 11
2.5. Componentes típicos de un sistema experto 16
2.6. Componentes mínimos de un sistema experto 17
2.7. Grafo de visibilidad 19
2.8. Diagrama de Voronoid 20
2.9. Decomposición en celdas 20
2.10. Potenciales artificiales 21

3.1. Diagrama de entrenamiento por refuerzo 22
3.2. Diagrama del agente mediante valor Q 23
3.3. Diagrama del agente mediante DQN 24
3.4. Diagrama inicial del agente empleando un sistema experto 25
3.5. Diagrama intermedio del agente empleando un sistema experto 25
3.6. Diagrama final del agente empleando un sistema experto 26
3.7. Diagrama de secuencia entrenamiento por refuerzo 27
3.8. Obtener acción Q value 28
3.9. Obtener acción de un DQN 29
3.10. Almacenar interacciones entre el agente y el entorno de entrenamiento. 29
3.11. Entrenamiento del agente DQN 30
3.12. Diagrama de secuencia entrenamiento por refuerzo propuesto 31
3.13. Obtener acción del agente propuesto 32
3.14. Almacenar datos del agente propuesto 33
3.15. Entrenamiento del agente propuesto 33
3.16. Sobrelape de categorías 34
3.17. Seccionamiento de datos 35

4.1. Entorno de entrenamiento 41

4.2. Variables que componen el estado 43
4.3. Distancia del vehículo al punto de destino 44
4.4. Diagrama de clases del sistema de entrenamiento 46
4.5. Simulaciones realizadas con el agente Random 47
4.6. Simulaciones realizadas con el agente DQN 47
4.7. Simulaciones realizadas con el agente doble DQN 48
4.8. Preprocesamiento 49
4.9. Segmentación 49
4.10. Simulaciones realizadas con el agente Experto 51
4.11. Simulaciones variando los parámetros del agente experto 52

A.1. Organización del código fuente. 60

Lista de tablas

4.1. Parámetros del entorno de entrenamiento 41
4.2. Clasificación del entorno de entrenamiento 42
4.3. Configuración de los puntos de prueba del vehículo 42
4.4. Acciones posibles 42
4.5. Estado del entorno 43
4.6. Estado preprocesado del agente experto 48
4.7. Codificación de los puntos de prueba del vehículo 50
4.8. Estado codificado y su acción correspondiente 50
4.9. Resultados entrenamiento 52

Capítulo 1

INTRODUCCIÓN

La generación de trayectorias es un aspecto clave en la implementación de la robótica móvil, dado que se encarga de asegurar el paso seguro desde el punto de partida hasta el punto de destino, procurando que la ruta sea la más corta posible. Este problema se puede resolver mediante redes neuronales.

En la actualidad el uso de aprendizaje por refuerzo se utiliza generalmente para realizar entrenamiento de redes neuronales, las cuales por su naturaleza no pueden ser auditadas por un ser humano ya que su conocimiento se almacena en un conjunto de parámetros de entrenamiento conocidos como pesos sinápticos; por esto se propone el reemplazo de la red neuronal por un sistema experto basado en reglas que almacena su conocimiento en un conjunto de estas que puede llegar a ser revisado y ajustado por un ser humano.

El sistema propuesto generará las trayectorias de manera reactiva en un espacio 2D entre el punto de origen y el punto de destino mediante un sistema experto basado en reglas. Dichas trayectorias se generarán a partir de los siguientes datos: mediciones de los puntos de prueba vinculados al vehículo, la orientación del vehículo con respecto al entorno, y la orientación del punto de destino con relación al vehículo. Una vez procesados los datos por el agente se determinará la acción a realizar por el vehículo de manera iterativa hasta alcanzar el punto de destino o el límite establecido de pasos.

Este proyecto se realizó en el grupo de investigación DIGITI(Sistemas Digitales Inteligentes) de la Facultad Tecnológica de la Universidad Distrital Francisco José de Caldas(Ministerio de Ciencia Tecnología e Innovación, 2019).

El documento está organizado de las siguente manera:

En el primer capítulo se muestran el planteamiento del problema y los objetivos a cumplir; en el segundo se exponen los conceptos utilizados o relevantes para el desarrollo; en el tercero se desarrolla la solución del problema; en el cuarto se evidencian los resultados obtenidos; por último, en el quinto se exponen las conclusiones obtenidas en el desarrollo de la presente investigación.

1.1. Planteamiento del problema

Al querer realizar mediciones de variables físicas, experimentos o toma de fotografías desde la estratosfera(Schlatter, 2009) o espacio cercano típicamente hasta una altura de 30Km se encuentran dos opciones de conseguirlo las cuales varian principalmente en su fase de ascenso. La primera consiste en cohetes sonda(Grupo Artax, 2011), esta consiste en el lanzamiento de un cohete a la estratosfera portando los instrumentos de medición requeridos, este método conlleva altos costos inherentes al diseño y construcción del cohete sin tener en cuenta los costos en logística y seguridad. La segunda radica en el uso de globos sonda como vehículos portadores de los instrumentos que realizaran las mediciones y/o los experimentos en la estratosfera con un costo relativamente bajo(20minutos, 2009; StratoCat, 2014). Dichos experimentos pueden ser realizados por individuos, colectivos o agencias espaciales.

La falta de un sistema de guia para estos globos sonda hace que naveguen a la deriva sujetos a la dirección y velocidad del viento, esto genera grandes posibilidades de pérdida y/o destrucción de la sonda en el aterrizaje. Los escenarios de pérdida o daños de la sonda pueden naturales o humanos, entre los primeros encontramos el aterrizaje en una zona de difícil acceso, la inmersión en un cuerpo de agua, el potencial ataque de animales, entre los segundos estaría su desmantelamiento, hurto o daño al desconocer su origen y función. Lo cual hace que la fase de recuperación de la sonda sea crítica tanto en terminos de desplazamiento como en tiempo, de manera que el escenario ideal es que se logre recuperar la sonda en el menor tiempo posible sin poner en riesgo al personal encargado. Otro posible riesgo a tener en cuenta es el de daños a terceros (Guerrero, 2013; Hillcox, 2014).

El grupo de investigación DIGITI realizó un proyecto denominado “Sonda de vuelo autónomo Sabio Caldas” con el objetivo de realizar la medición de algunas variables medioambientales y la toma de imágenes del espacio cercano en el año 2016(Casallas, Coronel y col., 2020; Casallas, Hernández y col., 2020), el proyecto contó con el apoyo de la Fuerza Aérea Colombiana(FAC) principalmente en temas logísticos, así como las instalaciones de la base Marandúa ubicada en Vichada para la fase experimental

del vuelo. En dicha fase se logró la medición de las variables medioambientales propuestas y la toma de imágenes, sin embargo, no se pudo recuperar la sonda luego del aterrizaje en parte debido al difícil acceso, ya que la base está rodeada de múltiples cuerpos de agua y de selva espesa. Por lo cual se identifica la necesidad de generar un sistema de navegación mediante el cual se asegure la recuperación de la sonda.

Por lo anterior, se propone la implementación un sistema de navegación basado en inteligencia artificial el cual pueda ser auditado y/o ajustado por un operador humano. Adicionalmente se propone que el entrenamiento del sistema de navegación se realice por aprendizaje por refuerzo.

El entrenamiento de un agente mediante aprendizaje por refuerzo se basa en la interacción entre un entorno de entrenamiento y el agente a entrenar. El agente es el encargado de almacenar el conocimiento generado de dicha interacción; este conocimiento está basado en tres datos: el estado, la acción realizada y la recompensa obtenida al aplicar dicha acción sobre el entorno(Sutton y Barto, 2018). El almacenamiento de este conocimiento depende del tipo de agente y puede realizarse en una tabla en la cual se almacenaría una tupla por cada una de las diferentes interacciones entre el entorno y el agente. Otra de las alternativas de almacenamiento está en los pesos sinápticos de una red neuronal, los cuales se irán actualizando conforme avanza el entrenamiento.

El conocimiento obtenido en el entrenamiento estará contenido en una tabla o en los pesos sinápticos de una red neuronal, así para el primer caso el análisis por parte de un humano tendrá una complejidad elevada que dependerá del tamaño de la tabla en la cual se almacenaron los datos(Russell y Norving, 2016). Para el caso de la red neuronal el análisis será imposible ya que funciona como un sistema de caja negra en el que sólo se pueden observar los pesos sinápticos que son un conjunto de coeficientes que no brindan mayor información por sí mismos(Goulding, 1991; Kartam y col., 1995; Poplin y col., 2018).

La generación de un sistema de entrenamiento por refuerzo que permita realizar a un humano la auditoría del conocimiento adquirido, permitirá establecer una explicación acerca del funcionamiento del agente al determinar la trayectoria. Con esto se evolucionaría de un sistema de caja negra como lo es una red neuronal en dónde el sistema funciona, pero no se tiene información acerca de cómo lo hace a un sistema de caja blanca.

1.2. Formulación del problema

Dado que uno de los inconvenientes del aprendizaje por refuerzo es el no poder interpretar el conocimiento adquirido por una red neuronal, se presenta como problema de investigación:
¿Cómo realizar el entrenamiento de un sistema experto mediante aprendizaje por refuerzo para la generación de trayectorias?

1.3. Objetivos

1.3.1. Objetivo general

Establecer un modelo de entrenamiento de un sistema experto mediante aprendizaje por refuerzo para la generación de trayectorias.

1.3.2. Objetivos específicos

- Determinar los componentes necesarios para realizar el entrenamiento de un sistema experto aplicado a la generación de trayectorias.
- Establecer las interacciones entre los componentes requeridos para el entrenamiento del sistema experto.
- Diseñar una metodología que permita generar conocimiento de la interacción entre el entorno y el agente.

Capítulo 2

MARCO DE REFERENCIA

2.1. Inteligencia artificial

La inteligencia artificial (IA) estudia el cómo lograr que las máquinas realicen tareas que por el momento son llevadas a cabo de mejor manera por seres humanos(Rich y Knight, 1994). Es uno de los más recientes campos de la ciencia e ingeniería. Su desarrollo comenzó poco después de finalizada la Segunda Guerra Mundial, el termino de inteligencia artificial como tal fue acuñado en 1956(Russell y Norving, 2016).

Se centra en el estudio de los agentes inteligentes, es decir, los dispositivos capaces de percibir su entorno y tomar decisiones sobre la acción a realizar para maximizar las posibilidades de cumplir su objetivo. El término inteligencia fue tomado a propósito, se quiso buscar una similitud entre la parametrización y el proceso de decisión de estos dispositivos, y las habilidades humanas de aprender, abstraer, razonar y decidir con el objetivo de solucionar problemas; sobre todo aquellos problemas en los que la dificultad es tan grande que la intuición humana se muestra como determinante a la hora resolver los mismos de la manera más óptima(Arbeiza Guerra, 2017).

El test de Turing, propuesto por Alan Turing en 1950, fue diseñado para proveer una definición operacional satisfactoria de inteligencia. Una máquina pasa la prueba si un interrogador humano después de realizar algunas preguntas escritas no puede determinar si las respuestas son de una persona o de una máquina(Russell y Norving, 2016).
La máquina debe poseer las siguientes capacidades:

- **Procesamiento de lenguaje natural(NPL)** para comunicarse satisfactoriamente en inglés.

- **Representación de conocimiento** para almacenar lo que oye o lee.
- **Razonamiento automático** para usar la información almacenada para responder preguntas y llegas a nuevas conclusiones.
- **Aprendizaje de máquina** para adaptarse a nuevas circunstancias y para detectar y/o extrapolar patrones.

Muchas disciplinas han contribuido en el desarrollo de la inteligencia artificial, con ideas, puntos de vista o técnicas: la filosofía hizo concebible la inteligencia artificial al considerar que la mente de alguna manera funciona mediante un proceso mecánico que opera con conocimiento codificado en algún lenguaje interno, y que el pensamiento puede usarse para elegir qué acciones tomar. Desde las matemáticas se proporcionaron las herramientas para manipular enunciados de certeza lógica, así como enunciados probabilísticos inciertos; también se sentaron las bases para comprender la computación y el razonamiento sobre algoritmos. La economía formalizó el problema de tomar decisiones que maximicen el resultado esperado para el que decide. La neurociencia reveló algunos datos sobre cómo funciona el cerebro y las maneras en que es similar y diferente a las computadoras. La psicología aportó la idea de que los humanos y los animales pueden considerarse máquinas procesadoras de información. Por su parte, los lingüistas demostraron que el uso del lenguaje encaja en este modelo. Las ciencias de la computación proporcionaron las máquinas cada vez más potentes que hacen posible las aplicaciones de inteligencia artificial. Por último, la teoría del control se ocupa del diseño de dispositivos que actúan de manera óptima sobre la base de la retroalimentación del entorno. Inicialmente, las herramientas matemáticas de la teoría de control eran bastante diferentes a las de la inteligencia artificial, pero los campos se están acercando(Russell y Norving, 2016).

Algunas de las aplicaciones actuales de la inteligencia artificial son:

- Vehículos autonomos
- Reconocimiento del habla
- Planeación logística
- Robótica
- Traducción automática

2.2. Redes neuronales

La mente humana es el modelo para construir máquinas inteligentes, una idea básica dentro de la inteligencia artificial es la de simular el funcionamiento de nuestro cerebro. Para esto se emula una neurona biológica como la mostrada en la figura 2.1, mediante una neurona artificial como aparece en la figura 2.2. En realidad, la idea de construir una máquina inteligente usando neuronas artificiales es algo en lo se ha trabajado durante mucho tiempo. Las nuevas arquitecturas para redes neuronales han recibido el nombre de arquitecturas "conexionistas", en su mayor parte estas arquitecturas no se han concebido para imitar el comportamiento del cerebro humano, sino más bien para recibir alguna inspiración acerca de hechos conocidos sobre el funcionamiento del cerebro(Rich y Knight, 1994; Singh y Lone, 2020).

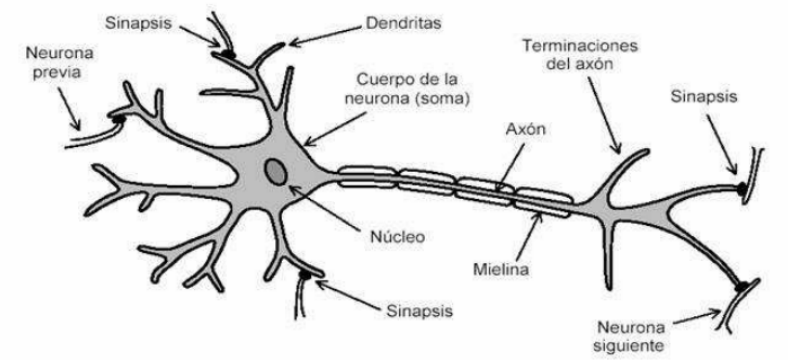


Figura 2.1: Neurona biológica(Mateo Jiménez, 2012).

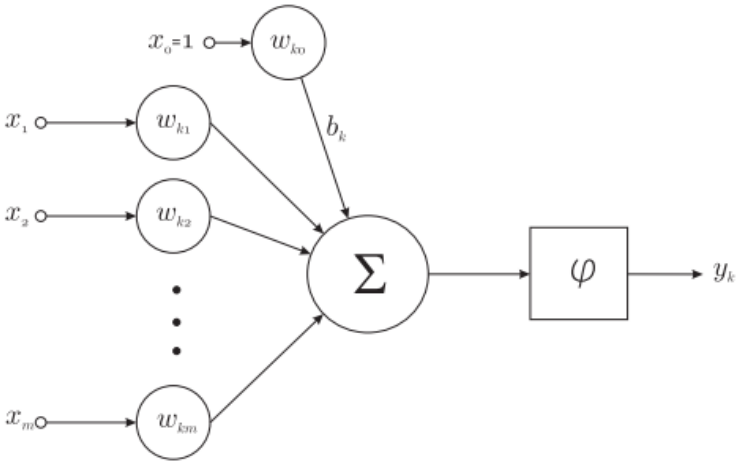


Figura 2.2: Diagrama de una neurona artificial(Mateo Jiménez, 2012).

La neurona biológica está compuesta del soma o cuerpo que contiene el núcleo celular. Del cuerpo se desprenden fibras llamadas dendritas y una única fibra de gran

longitud llamada axón. El axón tiene una longitud de aproximadamente un centímetro, lo cual constituye alrededor de 100 veces el diámetro del cuerpo. La comunicación entre neuronas se realiza mediante sus dendritas y axones, de manera que la neurona se emite a través del axón y recibe usando las dendritas. Esta comunicación se realiza mediante un proceso electroquímico a nivel de las sinapsis. Una neurona conecta con entre 10 y 100.000 neuronas. Las neuronas emiten señales que pueden ser excitadora o inhibidora de manera que activa o desactiva los receptores de la neurona que recibe la señal cuando se supera un umbral(Mateo Jiménez, 2012; Russell y Norving, 2016).

De manera análoga, en la red neuronal artificial se tiene un conjunto de entradas $\{x_1, x_2, ...x_m\}$ (estas señales pueden ser externas o de otras neuronas artificiales) que emulan las dendritas. Cada una de estas entradas tiene un peso asociado, cada neurona tiene un umbral. La suma ponderada de todas las entradas menos el umbral compone la activación de la neurona. Esta señal de activación pasa a través de una función de activación φ mediante la cual se producirá la salida de la neurona(Mateo Jiménez, 2012).

Entre los beneficios de las redes neuronales se pueden encontrar su gran capacidad de extraer significado, patrones o tendencias de datos complejos o imprecisos que son difícilmente detectables mediante otras técnicas computacionales o por un ser humano(Mateo Jiménez, 2012).

Adicionalmente:

- La habilidad de aprender mediante un conjunto inicial de datos sobre cómo realizar una tarea.
- Crea su propia organización y/o representación de la información durante el proceso de aprendizaje.
- Procesamiento en paralelo, lo cual diminuye el tiempo requerido para el procesamiento de los datos.
- La destrucción parcial de la red conlleva a una degradación de comportamiento, sin embargo, algunas de sus capacidades pueden permanecer a pesar del daño.

2.2.1. Topologías

Perceptrón

Es la topología más sencilla ya que sólo consiste de una neurona tal como se muestra en la figura 2.2. Es usado principalmente como un clasificador binario lineal. Tal como se indicó anteriormente las entradas x son multiplicadas por los pesos entrenados w, dichos pesos pueden ser entrenados mediante un algoritmo llamado backpropagation. La suma de la multiplicación de los pesos por el valor de la entrada es procesada por la función de activación, la que provee la salida de la neurona(Singh y Lone, 2020).

Perceptrón multicapa

El perceptrón multicapa es una red neuronal de tipo feedforward, esto implica que todo se propaga hacia adelante sin ningún tipo de retroalimentación. Se compone de una capa de entrada, una o más capas ocultas y una capa de salida tal como se observa en la figura 2.3. Es un modelo completamente conectado, lo que significa que una neurona se conecta con todas las neuronas de la capa anterior y de la capa siguiente. La capa de entrada no es realmente una capa de neuronas ya que no realiza ningún tipo de procesamiento, la capa de salida muestra las decisiones tomadas por la red neuronal, por esto existirán tantas neuronas como variables de salida sean requeridas(Mateo Jiménez, 2012).

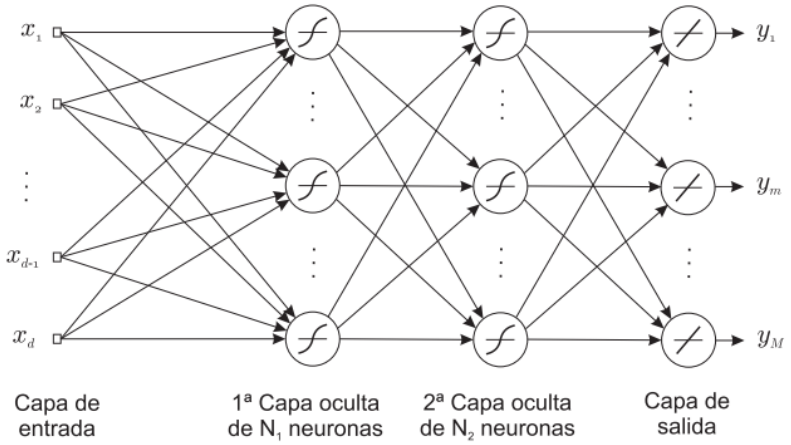


Figura 2.3: Perceptrón multicapa(Mateo Jiménez, 2012).

2.2.2. Funciones de activación

Lineal(Linear)

Consiste de una función que muestra a la salida el mismo valor de entrada, su rango de salida comprende desde -infinito a +infinito.

$$f(x) = x \tag{2.1}$$

Lineal rectificada(Rectified Linear Unit)

Es una función por partes en la cual la salida es igual a la entrada siempre y cuando esta sea positiva. En caso contrario la salida es cero. El rango de salida de la función va desde 0 a +infinito.

$$f(x) = \begin{cases} 0 \text{ if } x \leq 0 \\ x \text{ if } x > 0 \end{cases} \tag{2.2}$$

Sigmoide(Sigmoidal)

Se presenta como una función no lineal en la cual variaciones en la entrada realizan pequeños cambios en la salida. El rango de salida para esta función está entre 0 y 1.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

Tangente hiperbólica(hyperbolic tangent)

Tal como la función sigmoide consiste de una función no lineal, sin embargo, para esta función el rango de salida está entre -1 y 1.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.4}$$

2.2.3. Aprendizaje

Supervisado

Este es el entrenamiento más popular, parte de un conjunto de datos etiquetados con el valor esperado. Al realizar el entrenamiento se compara el valor esperado y el obtenido, la idea es que con el error se realice un proceso de actualización de los pesos de la red neuronal con el fin de obtener el valor esperado.

No supervisado

En este tipo de entrenamiento los datos no están etiquetados. Su funcionamiento se basa en técnicas de clustering o agrupamiento, su objetivo es agrupar muestras con características similares en el mismo cluster. La idea inherente de todas estas redes es que la capa oculta de neuronas debe ser capaz de extraer las características estadísticas del conjunto de datos de entrada. En la mayoría de los casos las neuronas de la capa oculta compiten entre ellas para determinar cuál está más próxima al dato de entrada, y dicha neurona es la que se actualiza, lo que significa que se mueve más cerca del dato de entrada(Mateo Jiménez, 2012).

2.3. Aprendizaje por refuerzo

El aprendizaje por refuerzo consiste en aprender qué hacer, qué acciones tomar para una situación específica en la cual se busca maximizar la recompensa recibida. Al alumno no se le dice qué acciones tomar, sino que debe descubrir qué acciones producen la mayor recompensa mediante ensayo y error. La idea básica es simplemente obtener los aspectos más importantes del problema real que enfrenta un agente en entrenamiento que interactúa a lo largo del tiempo con su entorno para lograr un objetivo, tal como se observa en la figura 2.4.

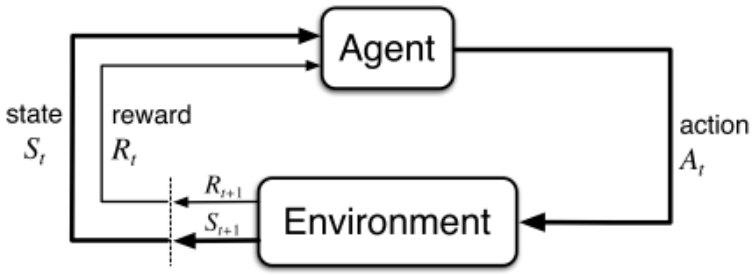


Figura 2.4: Interacción agente-entorno(Sutton y Barto, 2018).

El aprendizaje de refuerzo es diferente del aprendizaje supervisado, el tipo de aprendizaje estudiado en la mayoría de las investigaciones actuales en el campo del aprendizaje automático en el cual el entrenamiento se realiza mediante un conjunto de datos ya categorizados provistos por un supervisor externo(Sutton y Barto, 2018).

2.3.1. Elementos del aprendizaje por refuerzo

Entorno

Un entorno se puede clasificar de acuerdo a(Russell y Norving, 2016):

- Observabilidad
  Un entorno puede ser completa o parcialmente observable dependiendo del nivel de acceso a las variables de estado en cada punto de tiempo. Se dice que es completamente observable cuando se conoce toda la inforación relevante.

- Cantidad de agentes
  Como su nombre lo indica implica el número de agentes que pueden intervenir en el entorno. Cuando es multiagente puede ser competitivo o colaborativo.

- Deterministico o estocástico
  Si el siguiente estado está completamente determinado por el estado actual y la acción realizada es un entorno determinístico, en caso contrario es estocástico.

- Episódico o secuencial
  En un entorno episódico el agente recibe un estado y genera una acción, pero, cada episodio no depende de las acciones tomadas en episodios previos. En los secuenciales las acciones tomadas tendrán consecuencias a futuro.

- Estático o dinámico
  Indica si el entorno es modificado mientras el agente se encuentra determinando la acción a revisar, si se modifica entonces es dinámico, mientras que si espera la acción del agente para continuar entonces es estático.

- Discreto o continuo
  Esta característica aplica tanto al estado del entorno como al manejo del tiempo en las acciones del agente

Agente

Un agente en entrenamiento debe ser capaz de detectar el estado de su entorno hasta cierto punto y debe ser capaz de tomar medidas que afecten el mismo. El agente también debe tener una meta o metas relacionadas con el estado del entorno(Sutton y Barto, 2018).

Estado

Es el conjunto de valores de las variables observables de un entorno en un tiempo dado, dicho conjunto será usado por el agente para determinar una acción(Sutton y Barto, 2018).

Recompensa

Se define como el valor de una función que depende del estado del entorno y de la acción aplicada para modificar su estado, dicha función puede ser estocástica(Sutton y Barto, 2018).

2.3.2. Tipos de aprendizaje

Entre los tipos de aprendizaje por refuerzo pueden encontrarse:

Métodos basados en tablas

Estos métodos se aplican a problemas en los que el espacio de estado y de acciones es pequeño para las funciones de aproximación de valor que serán representadas como arreglos o tablas. En estos casos los métodos son capaces de encontrar soluciones exactas (Sutton y Barto, 2018). A continuación, una lista de algunos de ellos:

Q-Learning Se establece una tabla Q para almacenar las correspondencias entre el estado, la acción y la recompensa. El tamaño de la tabla depende del número de variables del estado y el número de acciones posibles.

$$Q_{target}(s,a) = r + \gamma maxQ(s',a') \tag{2.5}$$

Q_{target} representa la funcion de valor Q, donde r es la recompensa recibida, $\gamma \in [0,1]$ es el factor de atenuación que busca maximizar las recompensas futuras sobre las inmediatas. Por último $maxQ(s',a')$ representa el máximo valor Q para el estado s' y la acción a'.

Q-Learning doble En Q-Learning habrá un sesgo de maximización positivo al usar el máximo de las estimaciones como una estimación del máximo de los valores verdaderos. Supongamos que se divide el entrenamiento en dos conjuntos y se usa para aprender dos estimaciones independientes, que se establecerán como $Q_1(a)$ y $Q_2(a)$. Debe tenerse en cuenta que, aunque se tienen dos estimaciones, sólo se actualiza una de ellas al tiempo. Los requisitos en memoria se duplican, pero la capacidad de computo se mantiene. La actualización de la política viene dada por la siguiente ecuación:

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha[r + \gamma Q_2(s', argmaxQ_1(s',a)) - Q_1(s,a)] \tag{2.6}$$

$\alpha \in [0, 1]$ representa la tasa de aprendizaje, la cual se asegura de promediar la actualización teniendo en cuenta la aleatoriedad de las recompensas y transiciones de manera que converja alcanzando el valor óptimo de la función.

SARSA Al igual que con otros métodos Monte-Carlo, es un algoritmo que se enfrenta a la necesidad de la exploración y explotación. Lo primero que hace es aprender una función del valor de la acción en vez de una función de valor del estado. El algoritmo tiene una política inicial que se va actualizando al final de cada episodio. La regla de actualización de la politica es la siguiente:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \tag{2.7}$$

En la regla de actualización anterior, se observan las siguientes variables o elementos (s, a, r, s', a') lo cuál le da su nombre.

Métodos aproximados

Se constituye como una extensión de los métodos tabulares cuando se aplican a problemas con unos grandes espacios de estados. Muchos de los casos en los que se quiere utilizar aprendizaje por refuerzo, tienen un espacio de combinaciones muy grande. En estos casos no se puede esperar una política o función de valor óptima incluso con limites infinitos de datos y tiempo, por esto se busca encontrar una solución aproximada usando recursos computacionales limitados.

El problema con grandes espacios de trabajo no es sólo con la memoria requerida para almacenar las tablas con los datos, sino también con el tiempo necesario para llenarlas con datos precisos. En muchos de los casos cada uno de los estados a los que se llega no se habrá visto con anterioridad, lo que deriva en un hecho clave como es la generalización(Sutton y Barto, 2018). Entre los métodos aproximados se encuentran:

Deep Q-Network La correspondencia entre el estado, la acción y la recompensa es aprendida por una red neuronal mediante la aproximación de funciones(Riedmiller, 2005). Esto representa una mejora en comparación con Q-Learning, ya que este no tiene que realizar una búsqueda sobre una tabla, sino que realiza una evaluación de la red neuronal. Sin embargo, para realizar este entrenamiento se requiere un conjunto de datos obtenidos de las interacciones anteriores entre el entorno y el agente, por lo que debe crearse un espacio de almacenamiento para ello.

$$Q_{target}(s,a) = r + \gamma maxQ(s', a'; \theta_{target}) \tag{2.8}$$

θ_{target} representa los parámetros que serán modificados durante el proceso de aprendizaje.

Double Deep Q-Network De forma similar a como funciona Deep Q-Learning, la correspondencia entre el estado, la acción y la recompensa es llevada a cabo por una red neuronal. Sin embargo, ha sido probado que presenta problemas de sobreestimación debido a una función de aproximación con una flexibilidad insuficiente(Thrun y Schwartz, 1993), o por ruido(Hasselt, 2010, 2011), dado que puede ocurrir cuando los valores de la acción son incorrectos. Cuando esta sobreestimación tiene lugar afecta de manera negativa el desempeño, por ello, se ha realizado una mejora mediante la adición de una segunda red neuronal que tiene como objetivo corregir dicha sobreestimación(Hasselt y col., 2016).

$$Q_{target}(s,a) = r + \gamma Q(s', a' = argmaxQ(s', a', \theta_{eval}); \theta_{target}) \tag{2.9}$$

θ_{eval} representa los parámetros de la red neuronal que es usada como control sobre el problema de sobreestimación descrito con anterioridad.

2.4. Sistemas expertos

Los sistemas expertos resuelven problemas que normalmente son solucionados por "expertos" humanos. Para resolver problemas a nivel de experto, estos sistemas necesitan acceder a una importante base de conocimiento sobre el dominio que debe construirse de la manera más eficiente posible. También necesitan utilizar uno o más mecanismos de razonamiento para aplicar su conocimiento a los problemas que se le proponen. También necesitan un mecanismo para explicar a los usuarios que han confiado en ellos sobre lo que han hecho. Una forma de contemplar los sistemas expertos es que representan la mayor parte de la IA aplicada. Tienden a estar varios años retrasados respecto a los avances en la investigación(Rich y Knight, 1994).

Un sistema experto en inteligencia artificial se define como un programa informático que tiene la capacidad de representar y razonar sobre el conocimiento(Chojnacki y col., 2019).

2.4.1. Elementos de un sistema experto

En las figuras 2.5 y 2.6 se observan dos diagramas en lo que se exponen los componentes de un sistema experto, en estos se determina que los componentes mínimos requeridos son: el motor de inferencia, la base de conocimiento, mecanismo de aprendizaje y la interfaz de usuario(Tan, 2017).

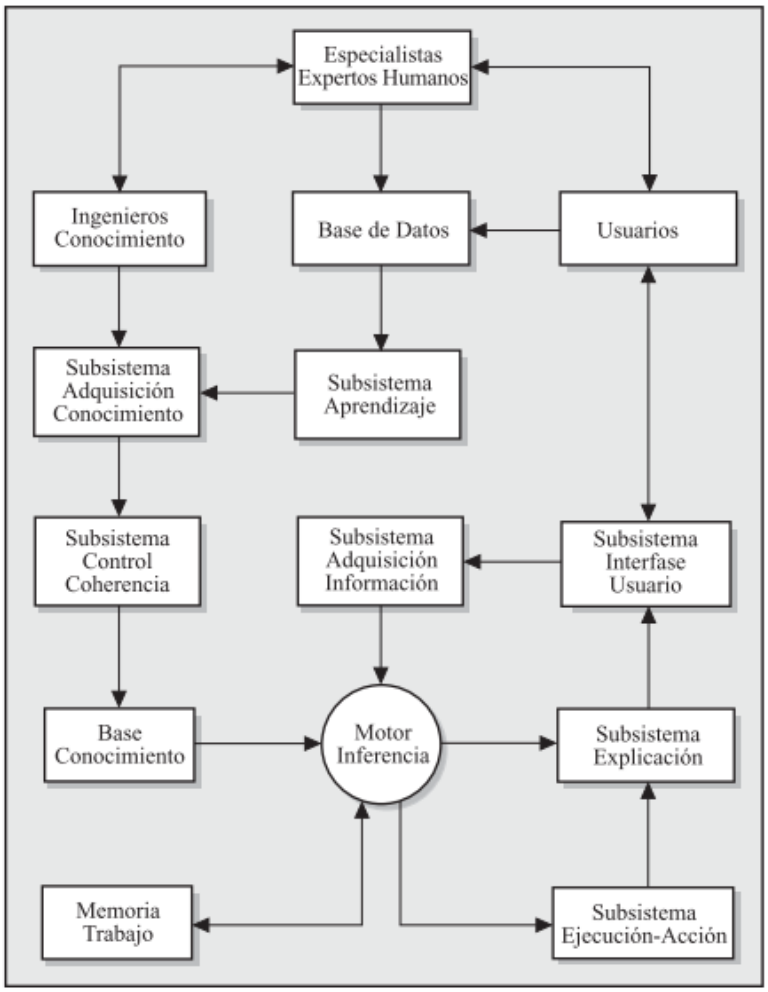


Figura 2.5: Componentes típicos de un sistema experto(Castillo y col., 1997).

Motor de inferencia

El motor de inferencia es el corazón de todo sistema experto. El cometido principal de este componente es sacar conclusiones aplicando el conocimiento a los datos. Las conclusiones del motor de inferencia pueden estar basadas en conocimiento determinista o conocimiento probabilístico(Castillo y col., 1997).

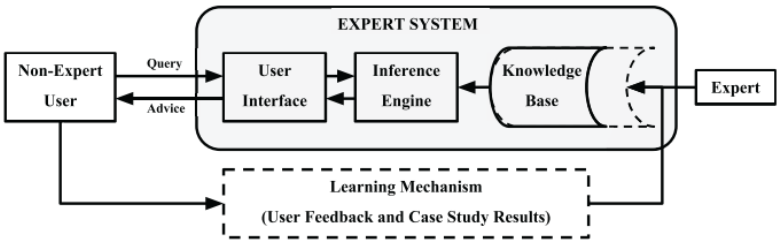


Figura 2.6: Componentes de un sistema experto(Ozden y col., 2016).

Base de conocimiento

Consiste en un conjunto de objetos(premisas o condiciones y conclusiones) y un conjunto de reglas que gobiernan las relaciones entre esos objetos. La información almacenada en la base de conocimiento es de naturaleza permanente y estática, es decir, no cambia de una aplicación a otra. Se debe diferenciar entre los datos y el conocimiento. El conocimiento se refiere a afirmaciones de validez general tales como reglas, distribuciones de probabilidad, etc. Los datos se refieren a información relacionada con una aplicación en particular(Castillo y col., 1997).

Reglas

En un sistema experto una regla se especifica como una acción o conjunto de acciones que serán realizadas en caso de presentarse una situación particular, que estará enmarcada en un conjunto de condiciones, por lo que una regla en muchos sentidos puede entenderse como un IF-THEN(Culbert y col., 2015; Gethsiyal Augasta y Kathirvalavakumar, 2012; Tan, 2017).

Hechos

Representa una pieza de información. En un sistema experto un hecho es la unidad fundamental usada por las reglas, una regla puede requerir la existencia de un conjunto de hechos o evidencias para generar un nuevo conjunto de hechos o conclusiones(Castillo y col., 1997; Culbert y col., 2015).

Mecanismo de aprendizaje

Controla el flujo del nuevo conocimiento que va del experto humano a la base de conocimiento. El sistema determina qué nuevo conocimiento se necesita, o si el conocimiento es realidad, es decir, si debe incluirse y en caso necesario incorporar dicho conocimiento(Castillo y col., 1997).

Interfaz de usuario

Se constituye como la interfaz entre el sistema experto y el usuario. Por ello para que sea efectiva debe incorporar mecanismos para mostrar y obtener información de forma sencilla y agradable(Castillo y col., 1997).

2.5. Generación de trayectorias

Como definición se trata de encontrar un movimiento libre de colisiones entre una configuración inicial y una final dentro de un entorno dado, es un tema geométrico al consistir en la ruta geométrica entre dos puntos. En aplicaciones de robótica avanzada, el problema de planeación de trayectorias es desafiante, especialmente para robots caracterizados por un gran número de grados de libertad, así como para robots que deben operar en entornos hostiles(Gasparetto y col., 2015).

De acuerdo a la forma de generar la trayectoria los métodos pueden ser denominados: deliberados o reactivos. La planeación deliberada es usada habitualmente en generación de trayectorias fuera de línea o en la replanificación en línea. La planeación reactiva por naturaleza opera completamente en línea, por lo cual no hay diferencia entre la planeación y el seguimiento de la trayectoria(Parry y Ordoñez, 2010).

Los algoritmos están usualmente divididos de acuerdo a la metodología empleada para generar la ruta geométrica:

- Técnicas de hoja de ruta
- Algoritmos de decomposición de celdas
- Métodos de potenciales artificiales

2.5.1. Técnicas de hoja de ruta

Se basan en la reducción de la configuración de un espacio N-dimensional a un conjunto de trayectorias unidimensionales en las cuales buscar, es decir, mapea el espacio libre en un sistema de curvas unidimensionales. Con lo cual la generación de la trayectoria consiste en la conexión de curvas entre el punto de origen y el punto de destino. Con esto es factible encontrar una ruta entre dichos puntos.

Grafos de visibilidad

Es común asociar un grafo a estas técnicas y generar una indexación de acuerdo a la optimización esperada, con lo cual el grafo puede ser escaneado en busca de la solución óptima para el problema de la generación de la trayectoria(Gasparetto y col., 2015).

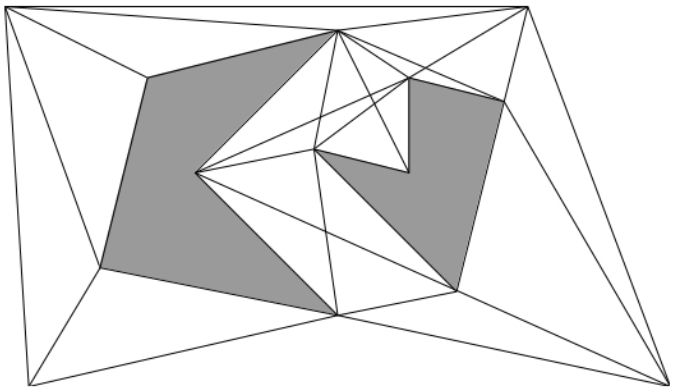


Figura 2.7: Grafo de visibilidad(Gasparetto y col., 2015).

En la figura 2.7 se muestra un grafo de visibilidad en el cual los nodos son los vértices de todos los obstáculos en la configuración del espacio. Los nodos representan puntos de ubicación mientras que las líneas representan las conexiones entre ellos. Los obstáculos que deben ser evadidos se muestran de color gris. El concepto de grafo de visibilidad que representa un hito en la generación de trayectorias fue introducido por Lozano-Pérez(Lozano-Pérez, 1983; Lozano-Pérez y Wesley, 1979).

Diagramas de Voronoi

Se define como una manera de dividir el espacio en regiones con las siguientes características, dado un conjunto de puntos $\{p_1...p_n\}$. Cada punto en la región i está más cerca al punto p_i que a cualquier otro(Gasparetto y col., 2015).

En la figura 2.8 se muestran once figuras geométricas que representan los obstáculos del entorno, mientras que las líneas verdes representan un conjunto de puntos equidistantes de al menos dos de los obstáculos. La línea morada representa la ruta generada. Las rutas generadas con esta técnica están diseñadas para pasar lo más lejos posible de los obstáculos.

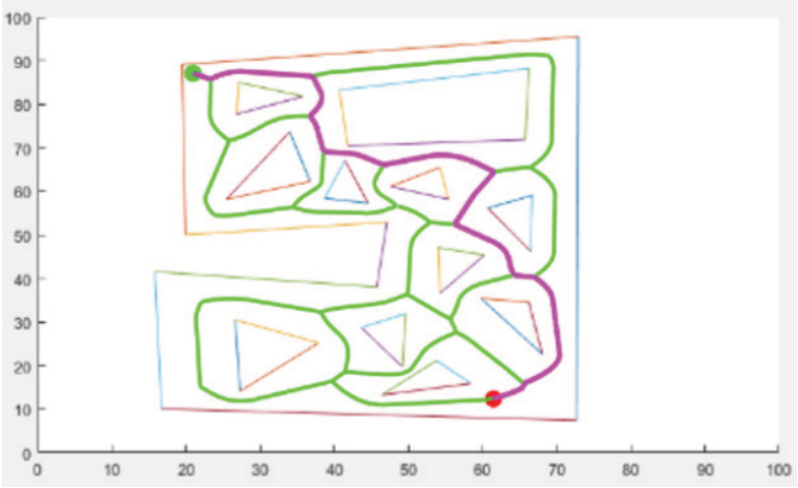


Figura 2.8: Diagrama de Voronoid(Alvarez y Flor, 2020).

2.5.2. Algoritmos de decomposición de celdas

En este método se realiza la subdivisión del espacio libre en un conjunto de celdas a partir de los vértices. Una vez generada la subdivisión se procede a crear un grafo en el cual los nodos representarán las celdas. Las conexiones entre ellos serán determinadas de acuerdo a la adyacencia de las celdas(Alvarez y Flor, 2020; Andrés y col., 2012; Gasparetto y col., 2015).

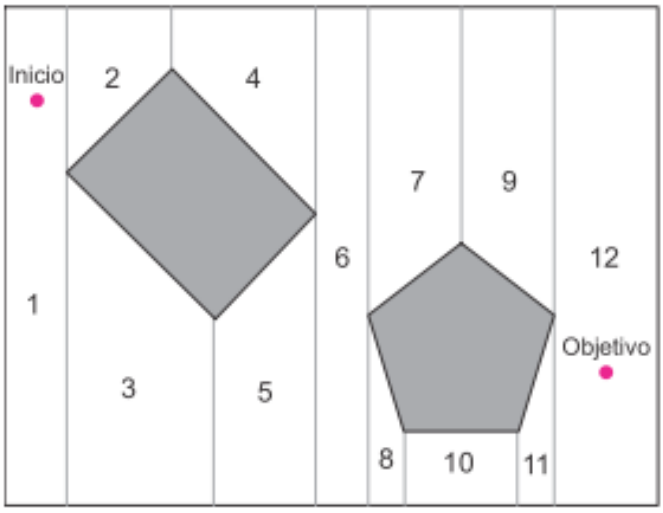


Figura 2.9: Decomposición en celdas(Andrés y col., 2012).

En la figura 2.9 se muestra una descomposición en la que se subdivide el espacio en doce celdas. En la celda número uno se ubica el punto de origen y en la celda doce el punto de destino.

2.5.3. Métodos de potenciales artificiales

Es un método que expresa la matemática de generación de trayectorias de forma simple, ya que es de fácil uso y muy eficiente. Este algoritmo funciona en tiempo real y tiene la capacidad de ser utilizado dinámicamente debido al sencillo tratamiento de obstáculos fijos y móviles. Este método se basa en una analogía realizada entre el desplazamiento de los robots móviles y los campos potenciales eléctricos, con la premisa de que el robot se desplaza como una partícula con carga eléctrica. El espacio libre por donde se mueve se considera un campo potencial, los obstáculos también tendrían una carga eléctrica, pero de igual signo al robot, de manera que entre ellos se repelen y el punto de llegada tendría carga eléctrica de signo contrario al robot, es decir, se atraen(Alvarez y Flor, 2020).

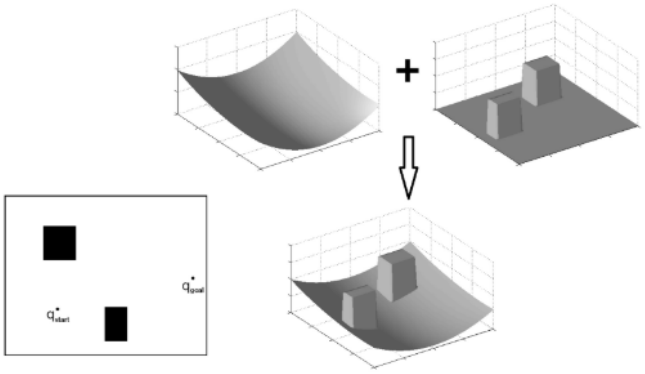


Figura 2.10: Potenciales artificiales(Gasparetto y col., 2015).

En la figura 2.10 se muestra un ejemplo en el cual existen dos obstáculos, un punto de inicio y un punto de destino. Adicionalmente se observa una pendiente entre el punto de inicio y el punto final.

Capítulo 3

DESARROLLO

3.1. Componentes necesarios para realizar el entrenamiento de un sistema experto aplicado a la generación de trayectorias

3.1.1. Componentes entrenamiento por refuerzo

Con el fin de determinar los componentes necesarios para el sistema propuesto se define una línea base descrita en la literatura tal como se muestra en la figura 3.1, en donde se observa la integración del entorno de entrenamiento y el agente que realiza el aprendizaje (Russell y Norving, 2016; Sutton y Barto, 2018).

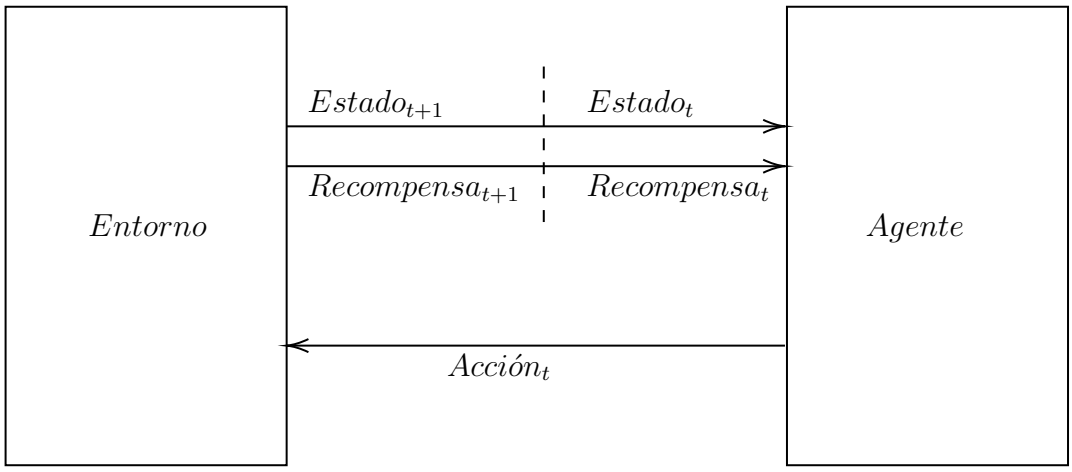


Figura 3.1: Diagrama de entrenamiento por refuerzo(Elaboración propia).

En las figuras 3.2, 3.3 se observan los diagramas de los agentes a entrenar los cuales tienen en comun dos componentes:

- Exploración / Explotación:
  Tiene como objetivo determinar si se realiza la explotación del conocimiento adquirido del agente o se realiza una exploración del entorno mediante la selección aleatoria de una acción.
  El método mediante el cual se determinará la acción se selecciona con base en un coeficiente que establece el factor de aprendizaje. El coeficiente tiene un valor inicial de uno con lo cual se asegura la exploración del entorno dado que la acción se determinará de manera aleatoria. Conforme avanza el entrenamiento el valor del coeficiente irá disminuyendo hasta un valor mínimo preestablecido, con lo cual se dará prelación a la explotación del conocimiento del agente sin que la selección aleatoria desaparezca por completo.

- Selección aleatoria:
  Se encarga de aportar un componente de exploración del entorno mediante la selección de una acción aleatoria que se ejecutará sobre dicho entorno.

En la imagen 3.2 se observan los componentes empleados en el entrenamiento por refuerzo mediante una función de valor Q. En dónde se observa que el conocimiento se almacena en la tabla Q y el agente se encarga de realizar una búsqueda en dicha tabla para obtener la acción a realizar de acuerdo al estado del entorno de manera que se maximice la recompensa.

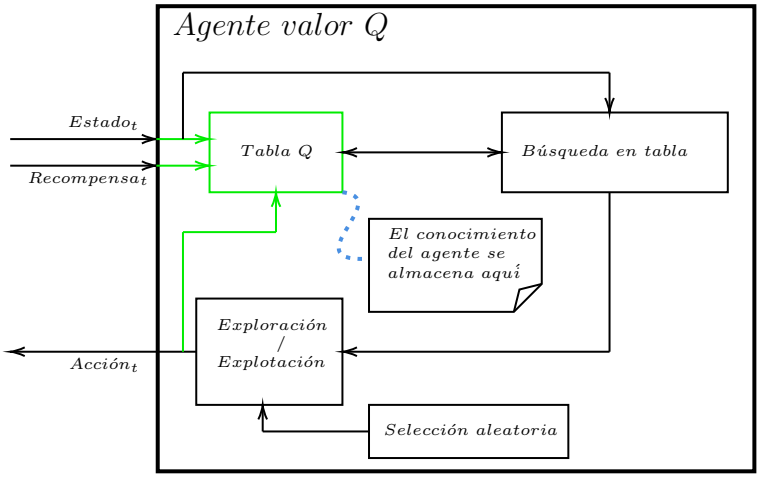


Figura 3.2: Diagrama del agente mediante valor Q(Elaboración propia).

En la imagen 3.3 se muestra como la tabla se reemplaza por un bloque de memoria con la misma función, almacenar las correspondencias entre el estado, la acción, la recompensa y el nuevo estado estableciendo un espacio finito en el número de registros a almacenar. En dicho espacio de memoria se almacena cada interacción entre el entorno y el agente. Los datos almacenados en dicho espacio son empleados para

realizar el entrenamiento de la red neuronal de manera que las entradas de las red neuronal deben encontrarse entre las variables del estado del entorno y la salida es la acción a realizar, de manera que el entrenamiento de la red neuronal maximice la recompensa.

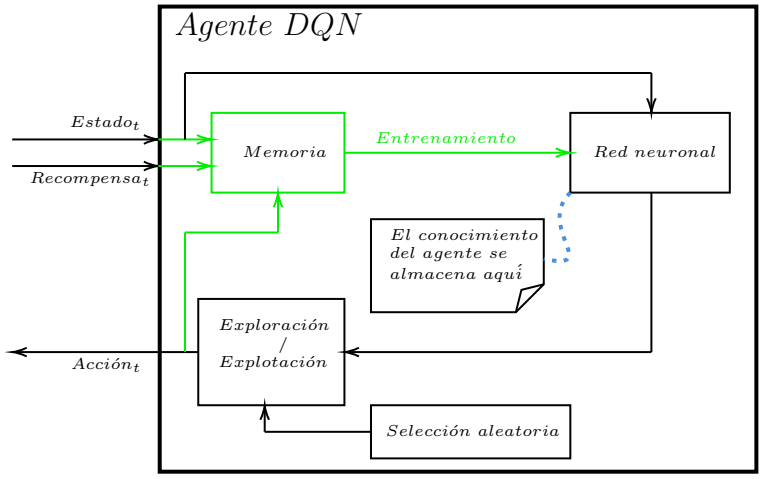


Figura 3.3: Diagrama del agente mediante DQN(Elaboración propia).

3.1.2. Componentes de entrenamiento por refuerzo de un sistema experto

Una vez realizada la consulta de los componentes empleados en diferentes metodologías de entrenamiento por refuerzo tanto para la función de valor Q como para redes de aprendizaje profundo(DQN), los cuales se mostraron en las figuras 3.2, 3.3 se realiza la primera versión del diagrama de componentes para el entrenamiento de un sistema experto basado en reglas, tal como se observa en la figura 3.4, para ello se parte con los componente básicos como lo son la generación de una acción de manera aleatoria, un componente que controlará la exploración del entorno o la explotación del conocimiento adquirido, la memoria que almacenará la correspondencia entre el estado, la acción, la recompensa y el nuevo estado, los componentes propios de un sistema experto como lo son el motor de inferencia que será el encargado de determinar la acción a realizar de acuerdo a las reglas y al estado percibido, la interfaz de usuario que será la encargada de obtener el estado del entorno y comunicar la acción a realizar y la base de conocimiento en donde será almacenado el conocimiento adquirido mediante un conjunto de reglas, así como un componente de extracción de reglas que se alimenta de los datos contenidos en la memoria.

Sin embargo, como se mencionará posteriormente en la sección 3.3.1, se hace nece-

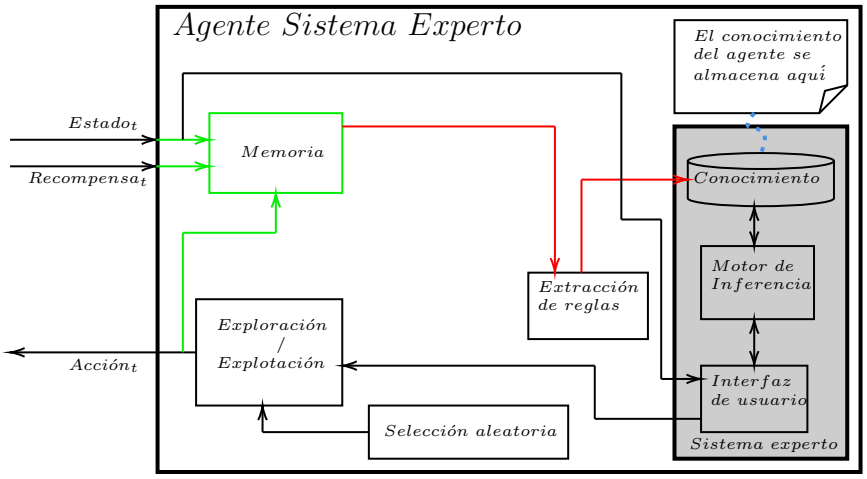


Figura 3.4: Diagrama inicial del agente empleando un sistema experto(Elaboración propia).

sario incluir un componente que aporte la capacidad de generalización de patrones a partir de datos aparentemente no relacionados, con lo cual se llega al diagrama de la figura 3.5 donde se observa la inclusión de una red neuronal y conexiones adicionales que le permitirán integrarse en el entrenamiento del sistema experto. El sistema de extracción de reglas ahora se apoyará en el conocimiento adquirido por la red neuronal de la siguiente manera: se tomarán todos los estados disponibles en memoria, utilizará la red neuronal para determinar la acción apropiada para dichos estados y realizará la extracción de las reglas basandose en dicha información, y por último cargará las reglas generadas en la base de conocimiento del sistema experto para que este las use para determinar la acción en función del estado.

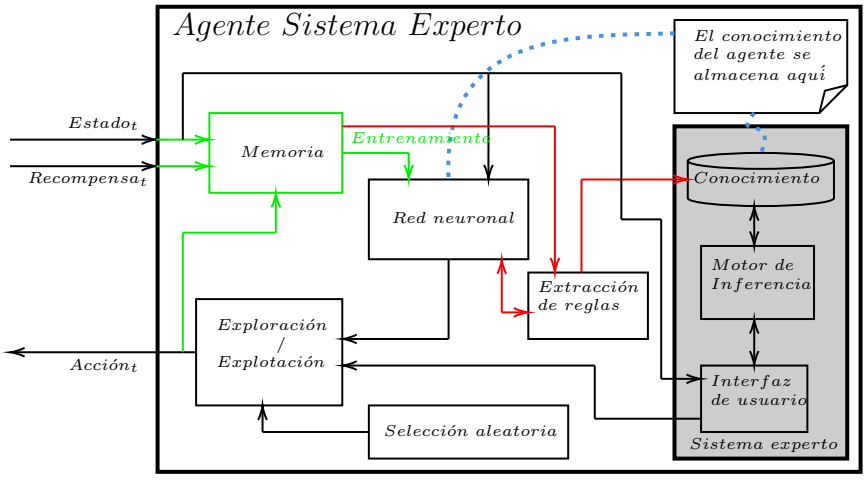


Figura 3.5: Diagrama intermedio del agente empleando un sistema experto(Elaboración propia).

Una vez analizados los resultados tal como se verá más adelante en la sección 3.3.1,

se hace necesario incluir un nuevo bloque encargado de realizar un preprocesamiento y codificación del estado antes de ser procesado por el sistema de extracción de reglas o por el sistema experto, con lo cual del diagrama final queda tal como se observa en la figura 3.6.

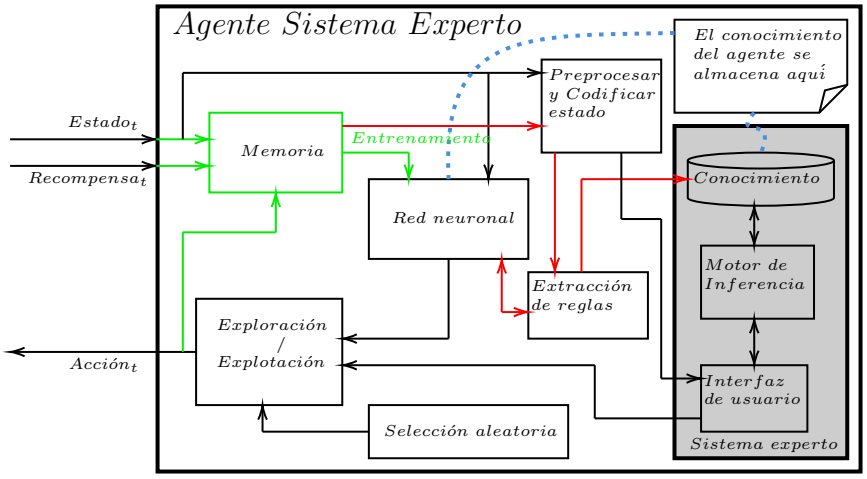


Figura 3.6: Diagrama final del agente empleando un sistema experto(Elaboración propia).

3.2. Interacciones entre los componentes requeridos para el entrenamiento del sistema experto

3.2.1. Interacción entre los componentes empleados para el entrenamiento por refuerzo

Una vez realizada la recopilación de los componentes involucrados se procede a establecer las interacciones entre los componentes involucrados para el entrenamiento por refuerzo observado en la figura 3.3, esta interacción se muestra en la figura 3.7. En la sección azul se muestran las interacciones realizadas por cada episodio del entrenamiento el cual se realiza hasta completar el número de episodios a realizar. En la sección verde se muestran la interacciones por cada paso realizado dentro de cada episodio, dicho episodio se termina una vez se alcance alguna de las siguientes condiciones:

- Se alcanzó el número máximo de pasos por episodios.
- Se alcanzó un estado terminal.

Adicionalmente se muestran tres bloques que realizan la diferenciación entre los diferentes tipos de entrenamiento:

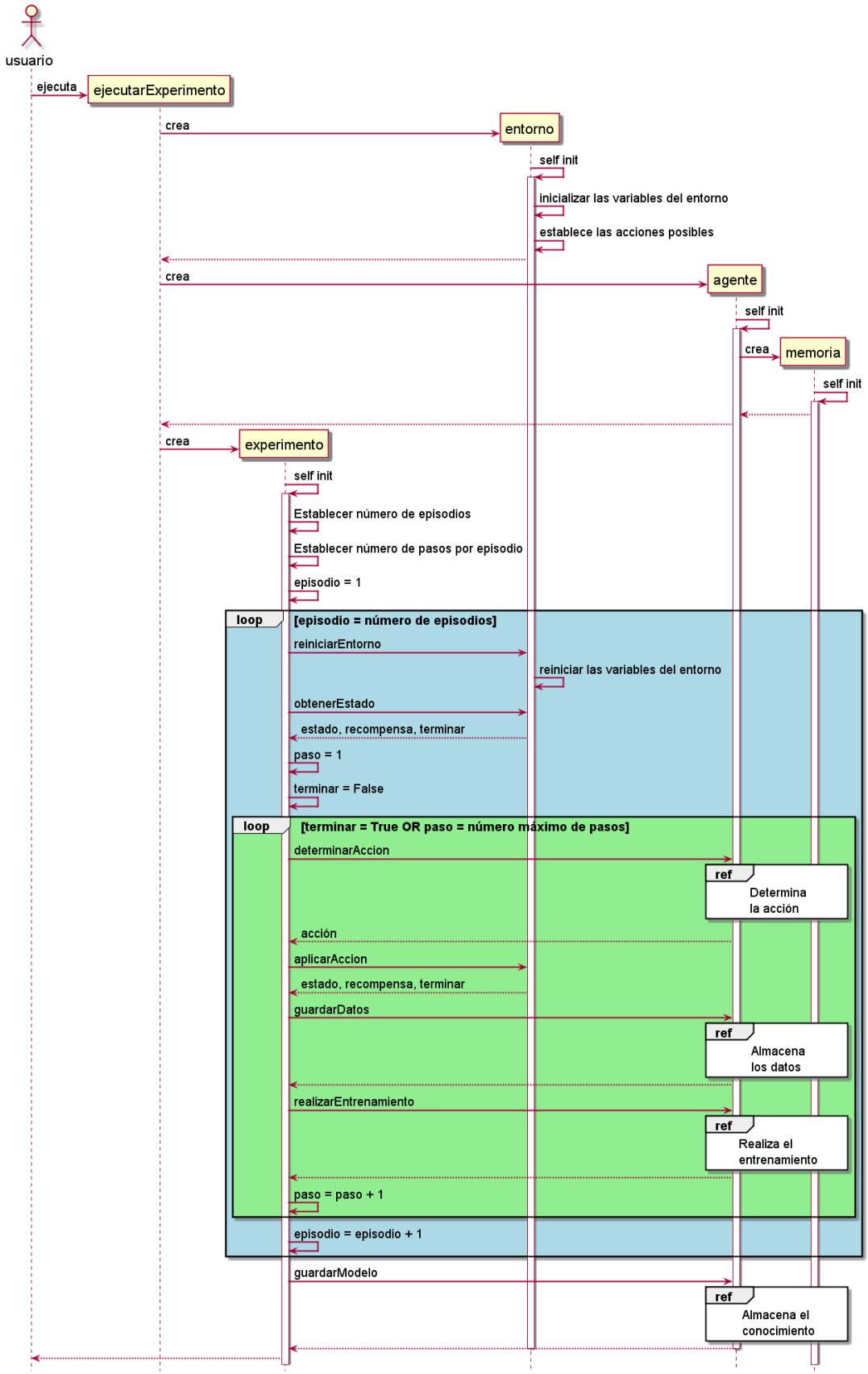


Figura 3.7: Diagrama secuencia de los componentes empleados en el aprendizaje por refuerzo(Elaboración propia).

- Determinar acción.
- Almacenar datos.
- Realizar entrenamiento.

Determinar acción

Este bloque tiene como función determinar la acción a realizar sobre el entorno. Al iniciar el entrenamiento determina la acción de manera aleatoria con lo cual se busca realizar una exploración o reconocimiento del entorno, con el objetivo de recopilar una gran cantidad de datos. Con los datos recopilados se realiza el entrenamiento del agente. Conforme el agente va siendo entrenado pasa de manera paulatina del modo de exploración al modo de explotación donde la acción se determina de acuerdo al conocimiento ya adquirido por parte del agente.

En la figura 3.8 se muestra la obtención de la acción por parte del agente de función de valor Q en dónde se realiza una consulta a la memoria con base a las variables del entorno.

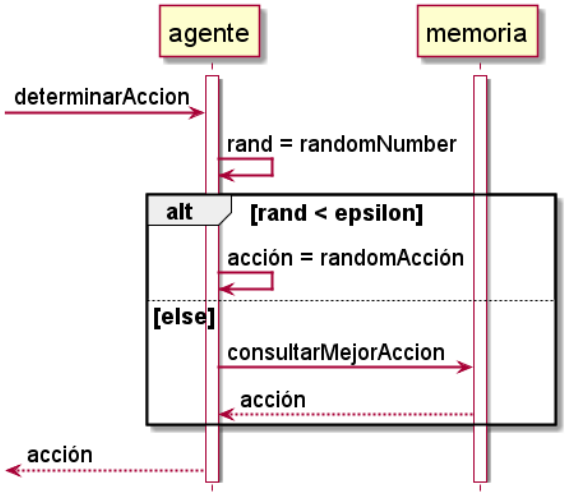


Figura 3.8: Determinar acción de un agente Q value(Elaboración propia).

En la figura 3.9 se muestra la obtención de la acción a realizar por parte del agente DQN en dónde se realiza una evaluación de la red neuronal, a la cual se le pasan las variables del entorno como parámetros de entrada.

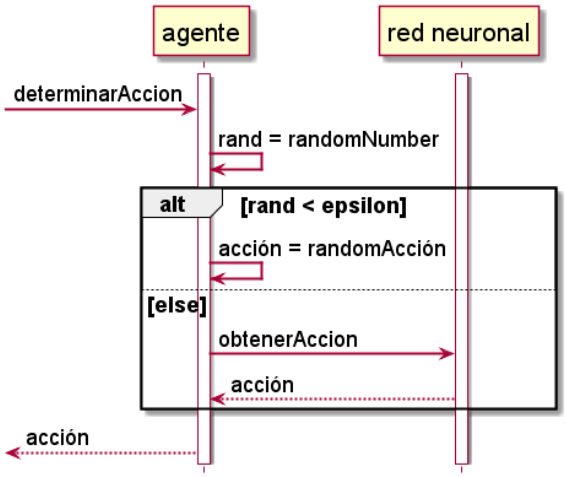


Figura 3.9: Determinar acción de una agente DQN(Elaboración propia).

Almacenar datos

Este bloque tiene como objetivo almacenar los datos que se requerirán para el entrenamiento del agente. Dicha operación se muestra en la figura 3.10 y eventualmente su diferencia radicará en la cantidad de datos que almacenará.

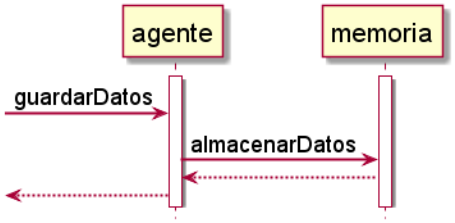


Figura 3.10: Almacenar interacciones entre el agente y el entorno de entrenamiento(Elaboración propia).

Realizar entrenamiento

Este bloque realiza el uso de los datos almacenados en la memoria para generar un ajuste en el conocimiento previamente adquirido.

En la figura 3.11 se muestra la interacción entre el agente DQN y su red neuronal. El agente consulta en la memoria almacenada los datos de las interacciones entre el agente y el entorno, los cuales se emplearán para realizar el entrenamiento de la red neuronal.

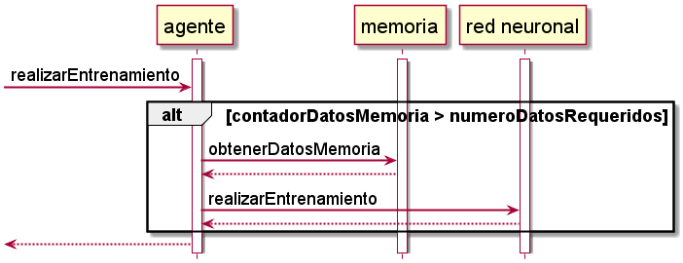


Figura 3.11: Entrenamiento del agente DQN(Elaboración propia).

3.2.2. Interacción entre los componentes empleados para el entrenamiento por refuerzo de un sistema experto

Con base en el diagrama mostrado en la figura 3.6 se debe establecer un mecanismo de aprendizaje mediante el cuál se realice la generación de las reglas que serán empleadas por el sistema experto para llevar a cabo la tarea propuesta. Con lo anterior en mente se parte del diagrama de secuencia mostrado en la figura 3.7 de manera que se relice un entrenamiento por cada paso una vez se alcance la cantidad mínima de datos para ello y un entrenamiento adicional por episodio tal como se muestra en la figura 3.12. En el entrenamiento realizado en cada paso el objetivo es entrenar la red neuronal que se encarga de proveer la generalización de los datos y el control de la recompenza esperada. Mientras que en el entrenamiento de episodio se realiza la extracción de las reglas para el sistema experto mediante la evaluación de la red neuronal con los datos contenidos en memoria.

Determinar acción

La manera en la cuál se determinará la acción a realizar se muestra en la figura 3.9. Se inicia generando acciones aleatorias hasta que se alcanza un número de datos suficiente para realizar el entrenamiento de la red neuronal. Una vez realizado el entrenamiento del sistema experto mediante la generación de las reglas se pasa a la explotación del conocimiento contenido en el sistema experto para determinar la acción a realizar tal como se muestra en la figura 3.13, sin que desaparezca el componente de exploración.

Almacenar datos

El procesamiento de almacenar los datos obtenidos no tiene mayores cambios a los mostrados en la figura 3.10. Se incluye un paso adicional con el objetivo de controlar el entrenamiento episódico del agente en el cual el objetivo es mantener las reglas de un agente que de buenos resultados. Para lo cual se emplea un factor de continuidad

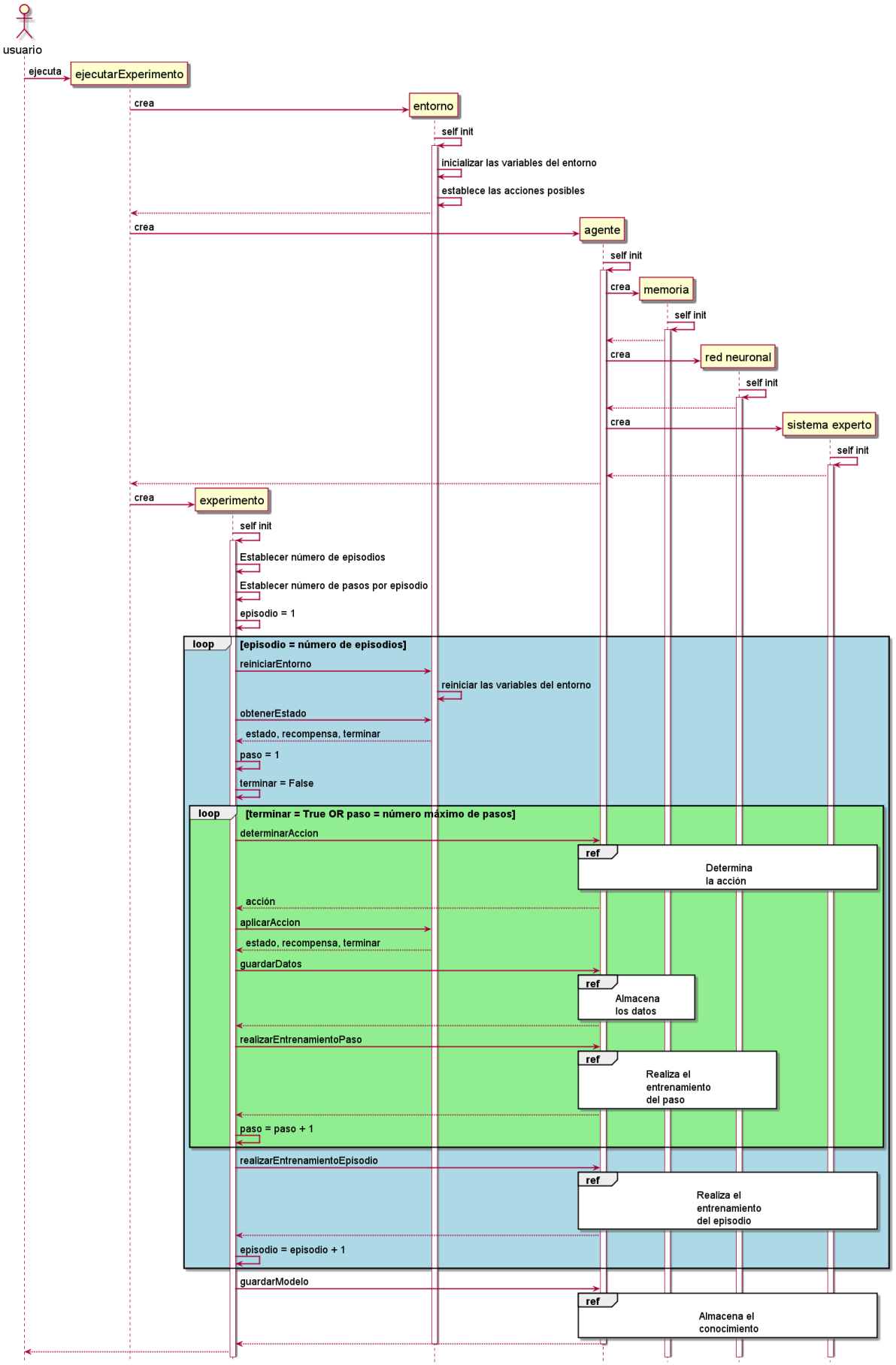


Figura 3.12: Diagrama secuencia de los componentes empleados en el aprendizaje por refuerzo de un sistema experto(Elaboración propia).

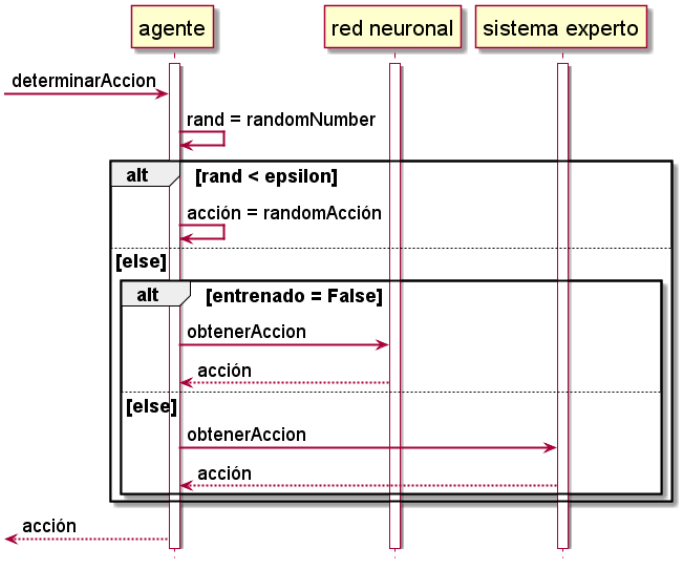


Figura 3.13: Determinar acción del agente propuesto(Elaboración propia).

que va decayendo conforme el agente resuelve de manera satisfatoria la tarea. El manejo del conteo de la continuidad del agente se muestra en la figura 3.14.

Realizar entrenamiento

Para realizar el entrenamiento del sistema experto mostrado en el diagrama de la figura 3.12 se evidencia que se requieren dos tipos de entrenamiento: el primero se encarga de realizar el entrenamiento de la red neuronal luego de cada paso y sería tal como el mostrado en la figura 3.11, el segundo se aplica sobre el sistema experto. En el segundo tipo de entrenamiento se realiza la consulta de los datos en memoria, estos serán preprocesados y codificados. Los datos serán consolidados en un solo conjunto, y pasarán a la etapa de extracción de reglas. El primer paso para la extración de las reglas es eliminar los datos duplicados, posteriormente serán clasificados por la red neuronal para determinar la acción a tomar en cada uno de ellos, y por último se procesan los datos para extraer un conjunto de reglas. Se realizará la asignación de dichas reglas al sistema experto, momento en el cual ya se pueden emplear para la toma de decisiones que modificaran el estado del entorno tal como se muestra en la figura 3.15.

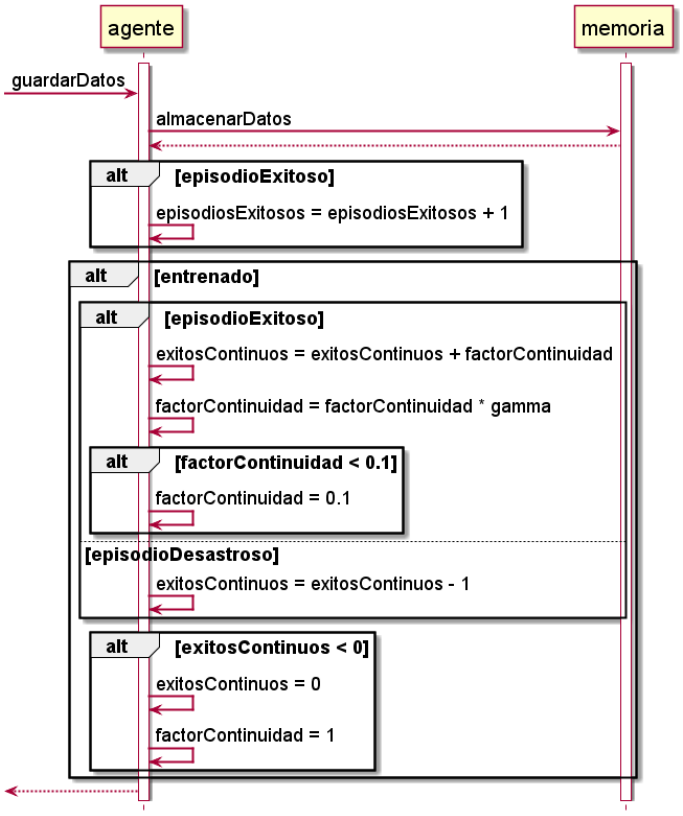


Figura 3.14: Almacenar datos del agente propuesto(Elaboración propia).

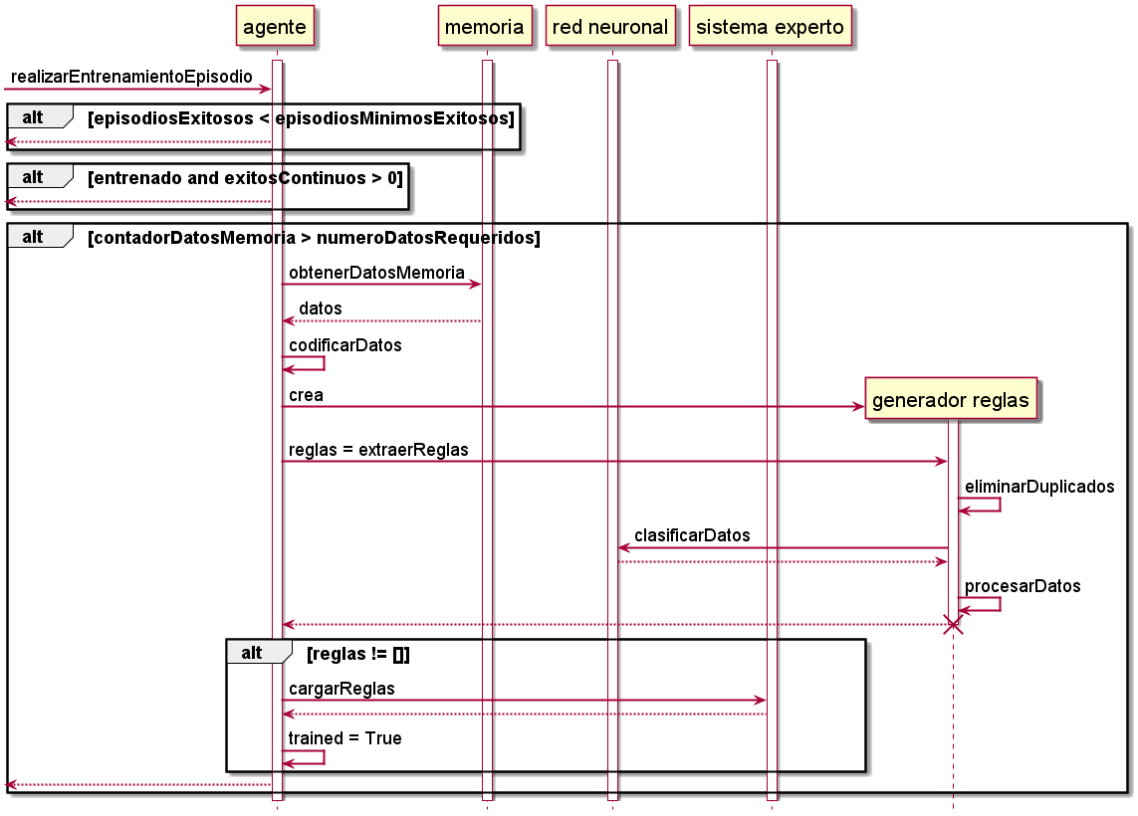


Figura 3.15: Entrenamiento del agente propuesto(Elaboración propia).

3.3. Diseñar una metodología que permita generar conocimiento de la interacción entre el entorno y el agente

3.3.1. Generación de reglas

Al momento de comenzar este proceso con los componentes mostrados en la figura 3.4 se observó que la extracción de reglas mediante arboles de decisión no genera los resultados esperados dada la poca generalización que se puede lograr ellos.

Con ello en mente se decide agregar al sistema una red neuronal que aportará la generación del conocimiento a partir de los datos contenidos en memoria tal como se muesta en la figura 3.5, para lo cual se empleó un método de extracción de reglas(Gethsiyal Augasta y Kathirvalavakumar, 2012; Vasilev y col., 2020) en donde se parte de una red neuronal entrenada y desde allí se determinan las entradas relevantes, posteriormente los rangos de datos relevantes por cada entrada en cada categoría y de esta manera se generan las reglas, dichas reglas deben ser organizadas iniciando por aquellas que involucran más variables de entrada de datos hacía las que involucran menos variables luego de esto se pasa por un proceso de poda en la cual se van eliminando condiciones que no influyen en la clasificación, por último se ajustan los rangos de datos de aquellas condiciones que puedan presentar un solapamiento con las condiciones de otras reglas para de esta forma ganar en precisión.

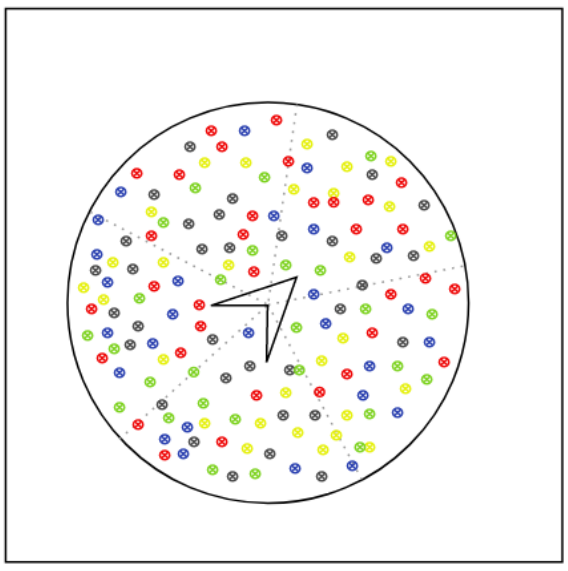


Figura 3.16: Sobrelape de categorías(Elaboración propia).

Sin embargo, al analizar los resultados obtenidos se determina que al generar las reglas de acuerdo a límites mínimos y máximos todas las reglas presentan un sobrelape en todas las variables de entrada, en la figura 3.16 se muestra a modo de ejemplo el sobrelape en las categorías al tomar mínimos y máximos para una de las varibles. Otro inconveniente con este proceso es que sólo permire una regla por cada categoría. Con ello en mente se optó por realizar una segmentación de los datos de entrada de acuerdo a la posición del punto de destino con relación al vehículo tal como se muestra en la figura 3.17 de esta manera se espera obtener un conjunto de reglas para cada sección.

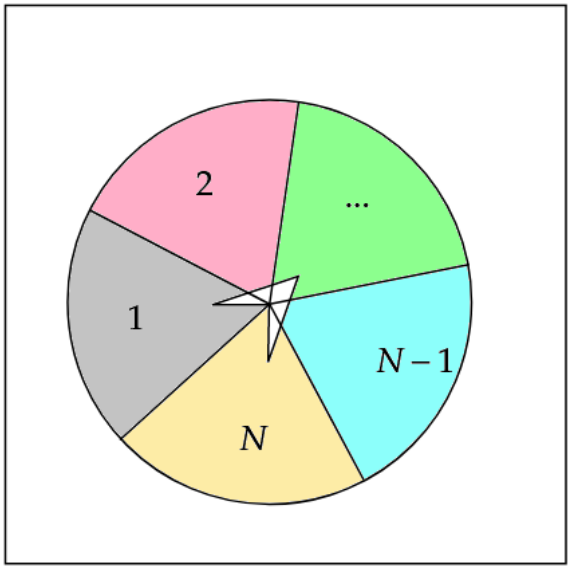


Figura 3.17: Seccionamiento de datos(Elaboración propia).

Al realizar pruebas y validar los resultados se observó que la implementación realizada no eran apropiada, ya que se debía eliminar una a una las entradas y asignar cero no era una opción ya que el cero es un valor válido en todas las variables de los estados, por lo cual se decidió agregar un paso de preprocesamiento tal como se muestró en la figura 3.6 en el cual realizar una codificación/discretización de los estados de manera que el cero fuera una opción no disponible en ninguna de las variables de estado. Sin embargo el sobrelapamiento en las condiciones de las reglas continuo generando reglas que no realizaban la tarea propuesta.

Algoritmo de generación de reglas

A continuación se muestran las definiciones y pasos mediante los cuales se realiza la generación o extracción de reglas empleadas por el sistema experto en la resolución de la tarea.

El estado del entono s se define como un vector con una lista de n propiedades sp,

$$sp = \{sp_1,\ sp_2,\ ...\ sp_n\} \tag{3.1}$$

una acción a se define como un vector de una propiedad ap_1,

$$ap = \{ap_1\} \tag{3.2}$$

dicha acción a se obtiene a partir de un estado s, el cual es procesado mendiante una red neuronal empleando la función de transformación definida aquí como $nn(s)$,

$$a = nn(s) \tag{3.3}$$

así mismo existe un estado codificado x, el cual se obtiene a partir de un estado s aplicando una función de codificación,

$$x = codificar(s) \tag{3.4}$$

el estado codificado x se define como un vector con una lista de m propiedades xp las cuales se calculan de acuerdo a la codificación,

$$xp = \{xp_1,\ xp_2,\ ...\ xp_m\} \tag{3.5}$$

existe un conjunto S que almacena mc estados s,

$$S = \{s_1,\ s_2,\ ...\ s_{mc}\} \tag{3.6}$$

posteriormente se define el objeto d como la unión de las propiedades del estado codificado x y la acción a

$$d = [x\ a] \tag{3.7}$$

con lo cual el objeto d se define como un vector con una lista de $m + 1$ propiedades

dp,

$$dp = xp \cup ap = \{xp_1,\ xp_2,\ ...\ xp_m,\ ap_1\} \tag{3.8}$$

los objetos d componen un conjunto D en el cual pueden repetirse los objetos d, por lo cual se requiere generar un conjunto DU en el cual los objetos d sean únicos, con lo cual el conjunto DU tendrá u elementos,

$$D = \{d_1,\ d_2,\ ...\ d_{mc}\} \tag{3.9}$$

$$DU \subset D \tag{3.10}$$

se establecen p propiedades de segmentación $segp$ para los datos del conjuto DU, dichas propiedades debe pertenecer a las propiedades del estado codificado xp

$$segp \subset xp \tag{3.11}$$

a partir del conjunto DU se obtienen k subconjuntos P mediante los valores de las propiedades de segmentación $segp$. Cada subconjuto P tiene ep elementos.

$$DU = \{P_1\ \cup\ P_2\ \cup\ ...\ \cup\ P_k\} \tag{3.12}$$

Se establecen las propiedades $agr1p$ mediante las cuales se realizará la agrupación de los elementos de cada uno de los conjuntos P, dichas propiedades son un subconjunto de las propiedades dp y deben excluir las propiedades de segmentación $segp$,

$$agr1p = \{agr1p_1,\ agr1p_2,\ ...\ agr1p_{m+1-p}\} \tag{3.13}$$

$$ap_1 \in agr1p \tag{3.14}$$

$$agr1p \in dp \tag{3.15}$$

$$agr1p \notin segp \tag{3.16}$$

como resultado de la agrupación se generan q subconjuntos AG, de cada subconjunto AG se generará un elemento g que tendrá una lista de propiedades gp, en dónde eg indica el número de elementos del subconjunto AG, los elementos g resultantes constituyen el conjunto G,

$$gp = \{agr1p,\ eg,\ min(segp_1),\ max(segp_1),\ ...\ min(segp_p),\ max(segp_p)\} \quad (3.17)$$

para determinar la regla predeterminada se establece la acción más común, se procesa una nueva agrupación del conjunto G mediante la propiedad ap_1 con lo cual se generan r conjuntos, para cada conjunto se genera un elemento f con las siguientes propiedades fp

$$fp = \{ap_1,\ fItem,\ fGrupo\} \quad (3.18)$$

en dónde $fItem$ es la sumatoría de la propiedad eg dividida entre el número total de elementos ep del conjunto,

$$fItem = \frac{\sum eg}{ep} \quad (3.19)$$

$fGrupo$ es el número de elementos g agrupados en el item f divididos sobre el total de grupos AG en el conjunto P.

$$fGrupo = \frac{elementos\ g}{q} \quad (3.20)$$

Posteriormente se calcula el promedio de $fItem$ y $fGrupo$ para cada elemento f. La acción del elemento f con el mayor promedio se establece como la acción por defecto, para lo cual se establecen las condiciones mínimas y máximas de los datos del conjunto P para las propiedades de segmentación $segp$.

Para determinar las reglas restantes eliminan los elementos g del conjunto G que tengan la misma acción que la regla por defecto, a continuación se itera sobre los elementos g del conjunto G organizados de manera descendente se acuerdo a la propiedad eg, se establecen las condiciones de acuerdo a los valores de las propiedades de agrupación $agr1p$ así como los valores mínimos y máximos de las propiedades de segmentación $segp$.

El proceso se muestra de manera ordenada en la siguente tabla:

Algoritmo 1: Extraer reglas

Data: datos almacenados en memoria.
Data: k =>número de particiones.
Data: nr =>número de reglas por partición.
Result: conjunto de reglas
Se procesan los estados por la red neuronal(Ecuación 3.3).
Se preprocesan y/o codifican los estados(Ecuación 3.4).
Se combinan los estados codificados con las acciones(Ecuación 3.7)
Se eliminan los duplicados(Ecuación 3.9).
Se establecen las propiedades de segmentación(Ecuación 3.11).
Se realiza la segmentación de los datos en k conjuntos(Ecuación 3.12).
Se recorren los k conjuntos P.
for $p = 1;\ to\ k$ **do**
    Se definen las propiedades de agrupación(Ecuación 3.13).
    Se determina la regla por defecto a partir del elemento f con mayor promedio entre $fItem$ y $fGrupo$
    La regla se determina de acuerdo a los valores mínimos y máximos de las propiedades xp usadas para la segmentación $segp$.
    Se eliminan los elementos g que coincidan con la acción por defecto.
    Se recorren los elementos g ordenados por la propiedad eg de forma descendente.
    for $r = 2;\ to\ nr$ **do**
        La regla se determina de acuerdo a:
        los valores de las propiedades xp usadas para la agrupación $agr1p$.
        los valores mínimos y máximos de las propiedades xp usadas para la segmentación $segp$.
        Se agrega la regla al conjunto de reglas.

Capítulo 4

RESULTADOS

A continuación se mostrará cómo se llega a la generación reactiva de trayectorias al aplicar la metodología mostrada en el capítulo 3 en un entorno de entrenamiento por refuerzo. El agente implementa una red neuronal y un sistema experto basado en reglas. Cómo resultado del entrenamiento se genera un archivo xml que contiene las reglas a emplear por el sistema experto, dicho archivo puede ser auditado y ajustado, modificando así la eficiencia del agente a la hora de resolver dicha tarea.

4.1. Entorno de entrenamiento

Definición del entorno

Un entorno de entrenamiento o pruebas que genera una superficie de trabajo a partir de seis parámetros tal como aparecen en la tabla 4.1. Cuatro parámetros determinan los límites de la superficie, otro determina la distancia a la cual se considera se alcanzó la posición de destino y otro para la longitud del paso del vehículo. Los valores de dichos parámetros se mantienen para todos los episodios de la simulación. Las acciones disponibles a realizar sobre el entorno por parte del agente para modificar la orientación del vehículo se muestrán en la tabla 4.4. Adicionalmente, para cada episodio se generan de forma aleatoria los punto de origen y destino así como las zonas de peligro o zonas prohíbidas, la clasificación del entorno puede observarse en la tabla 4.2.
En el entorno que se muestra en la figura 4.1 a modo de ejemplo, se pueden observar sus límites como las líneas de color negro que forman un cuadrado de coordenadas (0, 0), (10, 0), (10, 10) y (0, 10), adicionalmente se muestran las zonas de peligro como polígonos de color amarillo y las zonas prohíbidas como polígonos de color rojo, también se muestra la posición actual del vehículo denotada por el punto de color

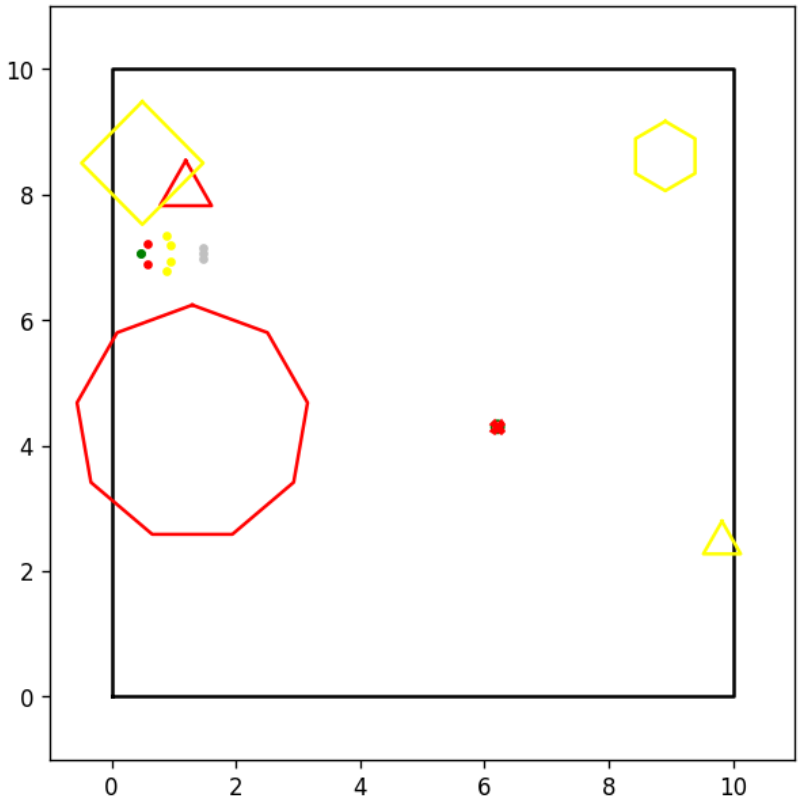


Figura 4.1: Entorno de entrenamiento(Elaboración propia).

Parámetro	Uso o significado
xmin	indica la coordenada inicial en el eje x
xmax	indica la coordenada inicial en el eje x
ymin	indica la coordenada inicial en el eje y
ymax	indica la coordenada inicial en el eje y
stepLength	indica la longitud que se desplazará en vehículo en cada paso.
targetTolerance	indica la longitud máxima entre el punto de destino y la ubicación del vehículo para determinar si este se alcanzó o no.

Tabla 4.1: Parámetros del entorno de entrenamiento.

verde, el punto de llegada el cual se muestra como una x de color rojo y finalmente los puntos de prueba vinculados al vehículo como dos puntos rojos, cuatro amarillos y tres grises en donde el color indica la distancía al vehículo.

Clasificación del entorno	
Observabilidad	Parcialmente observable
Cantidad de agentes	Único
Deterministico o estocástico	Deterministico
Estático o dinámico	Estático
Discreto o continuo	Discreto

Tabla 4.2: Clasificación del entorno de entrenamiento.

En la tabla 4.3 se muestra la configuración de los puntos de pruebas, unidades de longitud hace referencia al parámetro "stepLength" de avance del vehículo contenido en la tabla 4.1.

Alcance	Color	Ángulo [grados]	Distancia [unidades de longitud]	Lectura espacio libre	Lectura zona peligro	Lectura zona prohíbida
Corto (short)	Rojo	+/- 55	2	0	0.5	0.9
Medio (medium)	Amarillo	+/- 35 +/- 15	5	0 0	0.3 0.3	0.7 0.7
Largo (long)	Gris	+/- 5 0	10	0 0	0.1 0.1	0.5 0.5

Tabla 4.3: Configuración de los puntos de prueba del vehículo.

Acción	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Giro [grados]	-15	-10	-5	-3	-2	-1	-0.5	-0.2	-0.1	0	0.1	0.2	0.5	1	2	3	5	10	15

Tabla 4.4: Acciones posibles.

Estado u observación

El estado muestra información perceptible por el agente acerca del estado del vehículo, el punto de destino y las zonas de peligro o prohíbidas de acuerdo a las lecturas de los puntos de prueba, en la figura 4.2 se muestra gráficamente las variables que componen el estado, en la tabla 4.5 se muestra la informacion del estado en forma de vector.

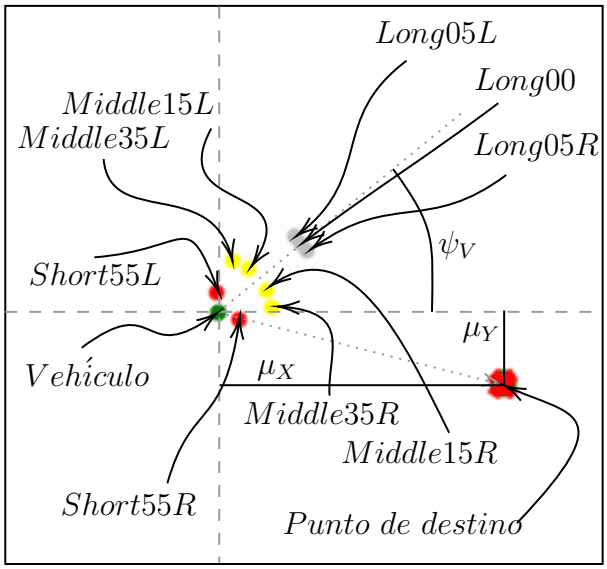


Figura 4.2: Variables que componen el estado(Elaboración propia).

μ_X	μ_Y	ψ_V	long 00	long 05R	long 05L	middle 15R	middle 15L	middle 35R	middle 35L	short 55R	short 55L

Tabla 4.5: Estado del entorno.

Las variables μ_X y μ_Y muestran las componentes del vector unitario de la orientación actual del punto de destino con relación al vehículo. ψ_V indica la orientación del vehículo con respecto al entorno. Los valores de las demás variables se toman de acuerdo a los datos mostrados en la tabla 4.3.

Recompensa

La recompensa se determina de acuerdo al estado del entorno así: si la posición del vehículo está incluida en una zona prohíbida o fuera de los límites del entorno la recompensa será de -10000, si la posición del vehículo se considera en el punto de destino la recompensa será de 10000, en caso contrario se realizará una sumatoría de los siguientes datos: -0.1 costo de cada paso; si la posición del vehículo está incluida en una zona de peligro será de -3, en caso contrario será de 0; los valores de las lecturas de los puntos de prueba según la zona en la cual se ubiquen dichos puntos(zonas libres, de peligro o prohíbidas) según la tabla 4.3.

Determinar el nuevo estado

El nuevo estado se determina a partir del estado actual y la acción que se realizará de acuerdo a la acción seleccionada por el agente de aquellas disponibles. El nuevo estado se determina a partir del estado actual, acto seguido se realiza un cambio

en la orientación del vehículo de acuerdo al giro seleccionado por el agente, luego el vehículo avanza una longitud determinada por el parámetro "stepLength", posteriormente se obtienen las lecturas de los puntos de prueba(ver tabla 4.3), las cuales consisten en indicar si es un espacio libre, una zona de peligro o una zona prohíbida. Una vez iniciado el episodio la posición del punto de destino así como las zonas de peligro o las zonas prohíbidas se mantienen constantes.

Finalización del episodio

Llegar al punto de destino

Para determinar si el vehículo ha alcanzado o no el punto de destino se realizan dos calculos tal como aparece en la figura 4.3, en el primer cálculo se determina la distancia euclidiana entre la posición actual y el punto de destino, posteriormente, en el segundo cálculo de determina la recta que forman la posición actual y la anterior, con base a esta recta se determina una recta perpendicular la cual debe pasar por la ubicación del punto de destino, finalmente se determina la distancia calculando la distancia euclidiana entre el punto de intersección de las rectas y la ubicación del punto de destino, una vez determinadas las distancias se toma la menor de ellas y se compara con el valor del parámetro "targetTolerance", si la distancia al punto de destino es menor o igual al valor del parámetro se da por alcanzado el punto de destino y finalizado el episodio.

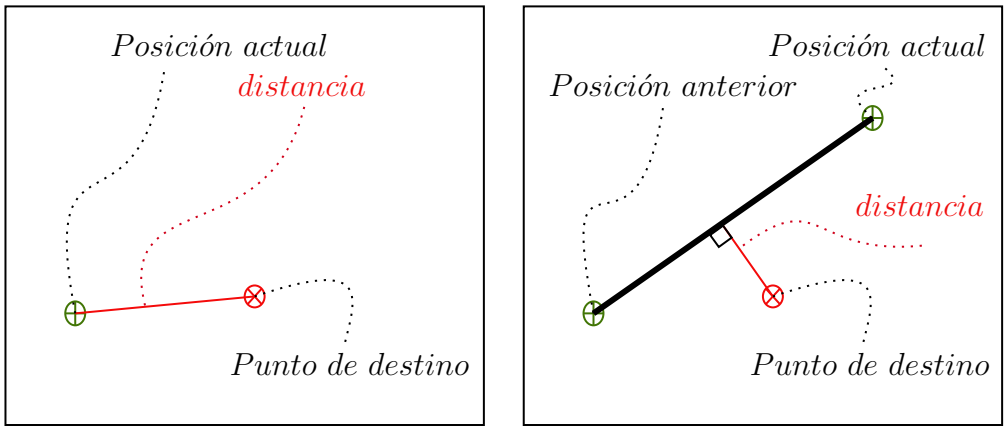


Figura 4.3: Distancia del vehículo al punto de destino(Elaboración propia).

Salir de los límites del entorno

Para determinar si el vehículo ha salido fuera de los límites del entorno se valida su posición con los valores máximos y mínimos del entorno que se encuentran en la

tabla 4.1. De manera si alguna de las componentes de la posición es menor al valor mínimo o superior al valor máximo se da por finalizado el episosio.

Ingresar a una zona prohíbida

Para determinar si el vehículo ingreso a una zona prohíbida se realiza el procesamiento mediante la ubicación del vehículo y los puntos que forman los polígonos de las zona usando el método de trazado de rayo para cada polígono(Haines, 1994; Perez-Carrasco, 2016; Sunday, 2012), si el vehículo esta contenido en alguno de dichos polígonos se da por finalizado el episodio.

4.2. Sistema de entrenamiento

Un sistema de entrenamiento por refuerzo mediante el cual se controla la comunicación entre el entorno y el agente seleccionado para la simulación. El entorno debe permitir la realización de dos operaciones como lo son el reinicio que tiene como misión llevar el entorno a un punto inicial para la simulación y el paso el cual debe recibir la acción señalada por el agente y realizar los cambios en el entorno para llevarlo al siguiente estado. Por su parte el agente debe proveer de al menos la operación de determinar la acción a realizar por parte del entorno para lo cual requerirá del estado actual del mismo. En la figura 4.4 se muestra el diagrama reducido de clases que permite realizar la simulación y intercambiar el agente.

En la implementación realizada se pueden generar diversos tipos de agentes como lo son Random, DQN, Doble DQN o Sistema experto. El agente de tipo Random sólo realiza la selección de una acción de manera aleatoria por lo cual no realizará ningún proceso de aprendizaje por lo cual no requiere memoria, pero sirve para validar el funcionamiento del sistema. Los agentes de tipo DQN y Doble DQN se basan en una red neuronal la cual puede variar de acuerdo a la topología de red elegida(Calderón y Camargo Casallas, 2020), también comparten la estructura de memoria que almacena las relaciones entre estado, acción y recompensa. La diferencia entre ellos como se mencionó anteriormente es que el agente de tipo DQN funciona con una red neuronal mientras que el Doble DQN funciona con dos redes.

En la figura 4.5 se muestran los resultados obtenidos luego de ejecutar la simulación por 2000 episodios, en la cual se puede observar en la parte superior la recompensa lograda por cada iteración y en la parte inferior la longitud de pasos realizados por cada iteración. En la gráfica los datos de cada iteración se muestran de color gris y de color azul se muestra la curva suavizada para que sea más fácil de observar la

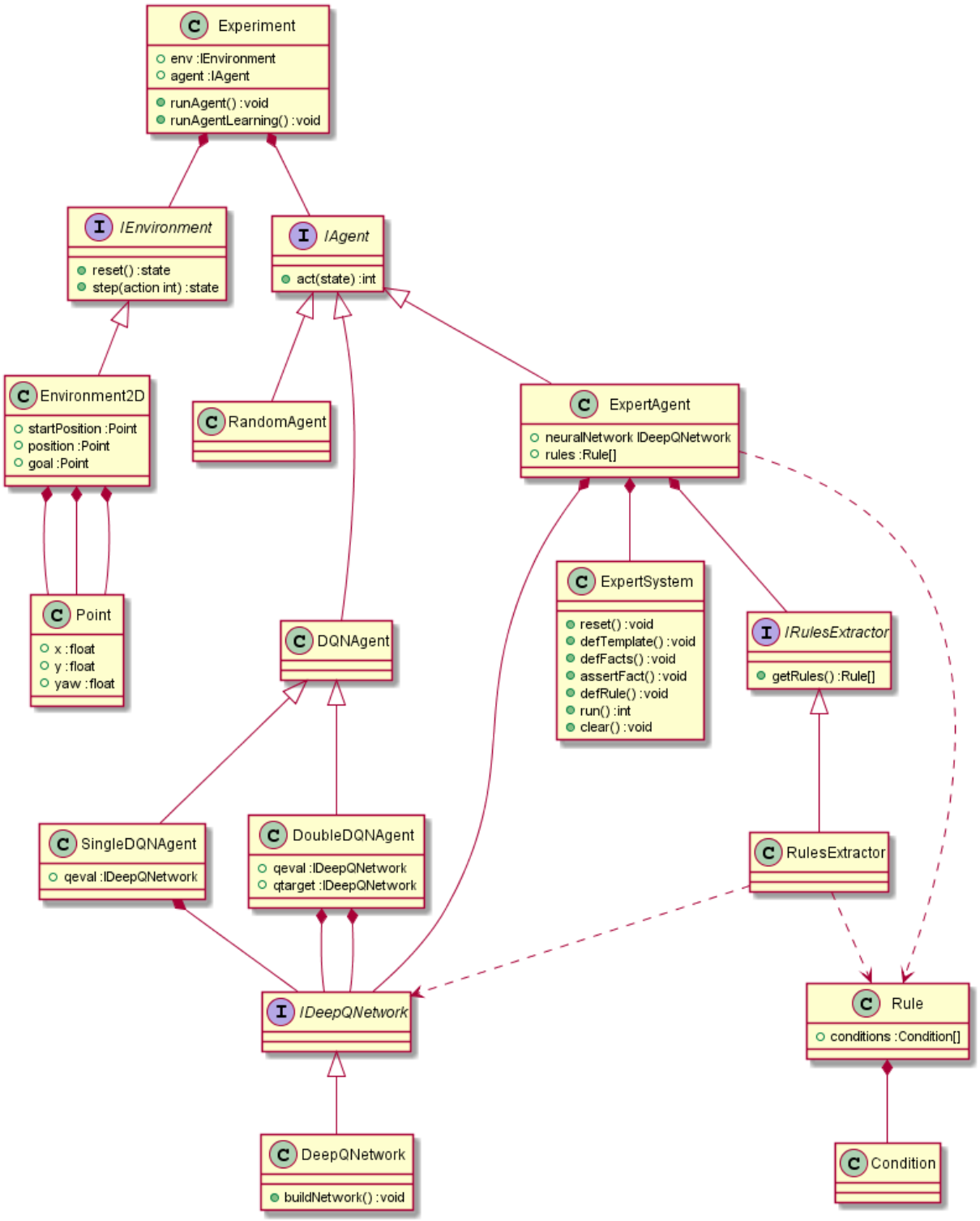


Figura 4.4: Diagrama de clases del sistema de entrenamiento(Elaboración propia).

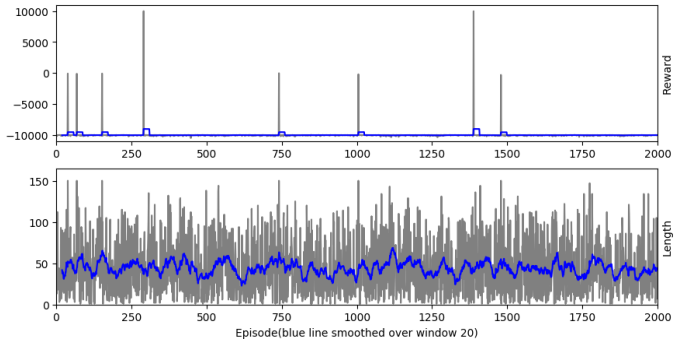


Figura 4.5: Simulaciones realizadas con el agente Random(Elaboración propia).

tendencia.

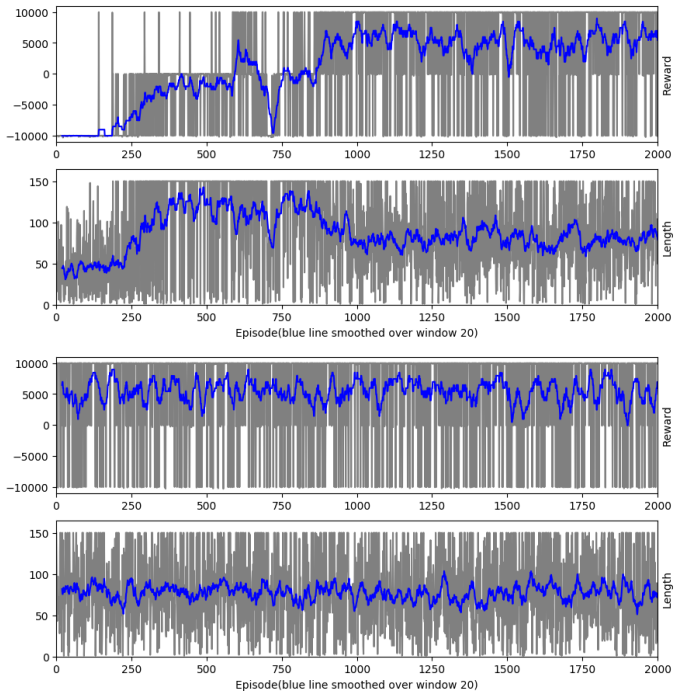


Figura 4.6: Simulaciones realizadas con el agente DQN(Elaboración propia).

En las figura 4.6 se muestran los resultados de entrenamiento en la parte superior mediante dos gráficas y en la parte inferior se muestra el resultado obtenido con el agente previamente entrenado, el agente logró generar la ruta en 1414 episodios 70.7 %, de igual manera la figura 4.7 muestra la misma información para el agente doble DQN, logrando generar la ruta en 1437 episodios 71.85 %.

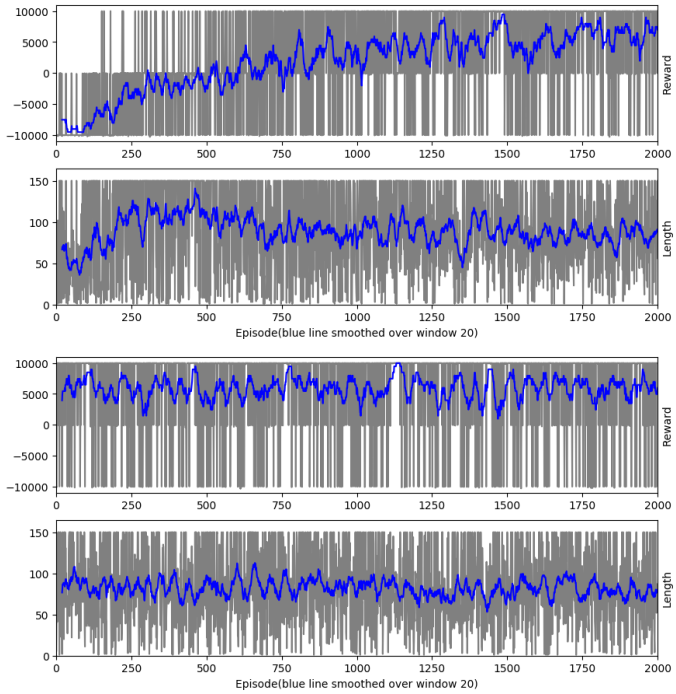


Figura 4.7: Simulaciones realizadas con el agente doble DQN(Elaboración propia).

4.3. Modelo de entrenamiento

En la etapa de preprocesamiento del diagrama propuesto para el agente experto de la figura 3.6 se realiza mediante el cálculo de la orientación del punto de destino con relación al vehículo a partir de las variables μ_X y μ_Y, una vez obtenido dicho ángulo se procesa con la variable ψ_V con lo cual se determina δ_{yaw}. δ_{yaw} se define como el ángulo que debe girar el vehículo para dirigirse al punto de destino tal como se observa en la figura 4.8.

Con lo cual la representación del estado que se usará para la generación de reglas es el indicado en la tabla 4.6.

δ_{yaw}	long 00	long 05R	long 05L	middle 15R	middle 15L	middle 35R	middle 35L	short 55R	short 55L

Tabla 4.6: Estado preprocesado del agente experto.

Las propiedades del estado de acuerdo a la ecuación 3.5 son:

$$xp = \{\delta_{yaw}, long00, long05R, long05L, middle15R, middle15L, middle35R, middle35L, short55R, short55L\} \tag{4.1}$$

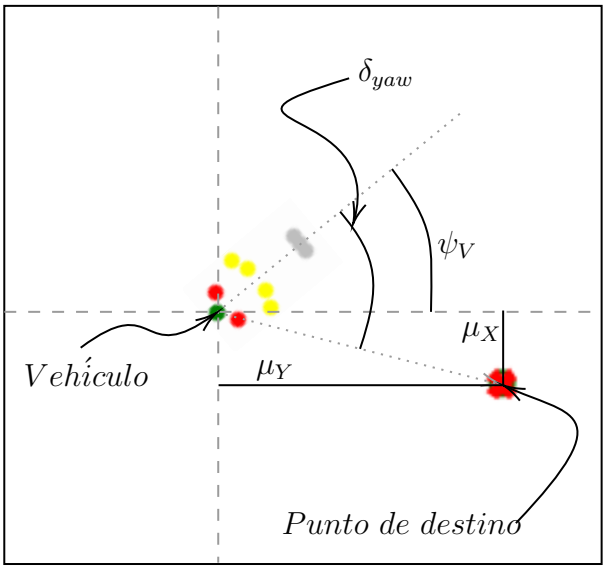


Figura 4.8: Preprocesamiento(Elaboración propia).

La codificación de los datos se presenta como una discretización de los datos, la cual se realiza dependiendo la variable a codificar, por ejemplo para la codificación de la variable δ_{yaw} se divide la circunferencia en 3600 segmentos iguales desde $-\pi$ hasta π tal como se muestra en la figura 4.9 con lo cual cada grupo tiene un ángulo de 0,1° ó de $2\pi/3600$, debido a esto la variable δ_{yaw} pasa a ser un entero entre 1 y 3600.

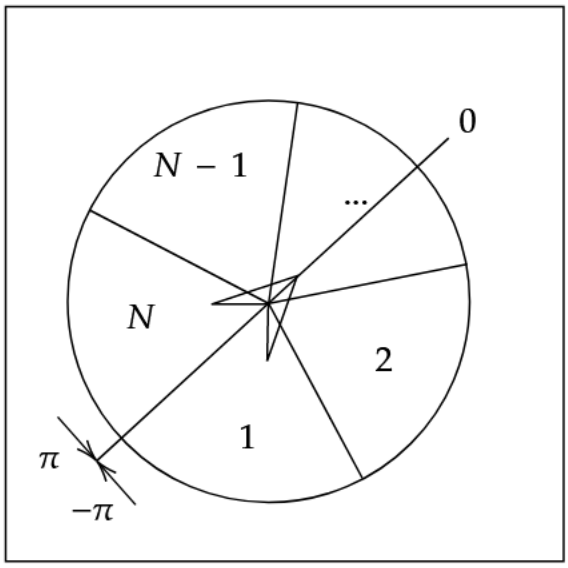


Figura 4.9: Segmentación(Elaboración propia).

Para los puntos de prueba la codificación se basa en los valores de las zonas indicados en la tabla 4.3 y se realiza de la forma que se indica en la tabla 4.7.

Tal cómo se indicó en las figuras 3.6, 3.15 y 4.4 en dónde se observa respectiva-

Lectura	Valor codificado
Espacio libre	1
Zona peligro	2
Zona prohíbida	3

Tabla 4.7: Codificación de los puntos de prueba del vehículo.

mente un diagrama de las partes que componen el sistema, así como un diagrama de secuencia del entrenamiento por episodio y el diagrama de clases que soporta el desarrollo de la presente investigación se muestra la iteracción de la red neuronal con el sistema de extracción de reglas en las cuales se basará el sistema experto para su misión de resolver el problema propuesto.

Una vez realizada la codificación de los estado así como su clasificación se obtiene un conjunto de datos con las propiedades que se observan en la tabla 4.8, del conjunto de datos se eliminan los datos duplicados.

δ_{yaw}	long 00	long 05R	long 05L	middle 15R	middle 15L	middle 35R	middle 35L	short 55R	short 55L	action

Tabla 4.8: Estado codificado y su acción correspondiente.

Con lo cual se establece que las propiedades de acuerdo a la ecuación 3.8 son:

$$dp = \{\delta_{yaw}, long00, long05R, long05L, middle15R, middle15L, middle35R, middle35L, short55R, short55L, action\} \tag{4.2}$$

Posteriormente, se realiza una segmentación de los datos de acuerdo a la propiedad δ_{yaw} de manera similar como se realizó su codificación. Por cada grupo se genera un conjunto de máximo 60 reglas tal como se mencionó en la sección 3.3.1.

$$segp = \{\delta_{yaw}\} \tag{4.3}$$

Así mismo se establecen las propiedades de agrupación:

$$agr1p = \{long00, long05R, long05L, middle15R, middle15L, middle35R, middle35L, short55R, short55L, action\} \tag{4.4}$$

En la figura 4.10 se muestra el resultado del entrenamiento de un agente experto por 2000 episodios, el sistema experto genera una segmentación de datos en 9 grupos

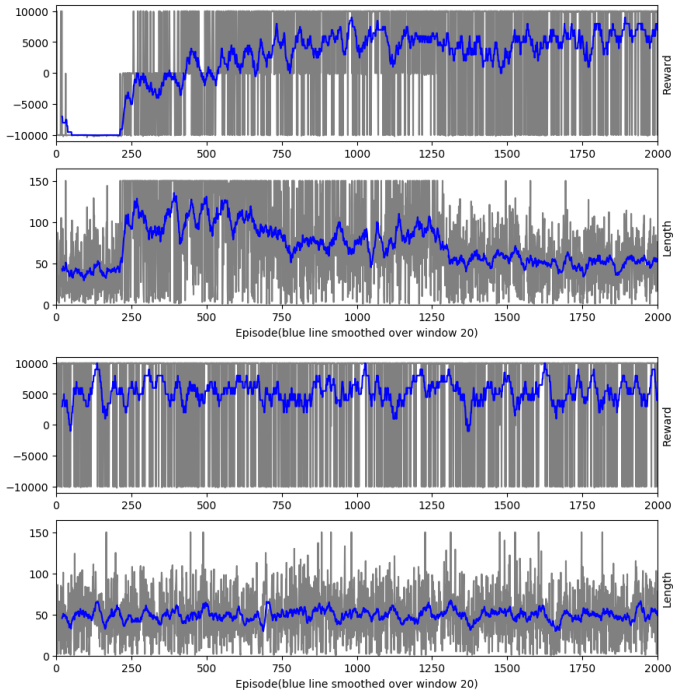


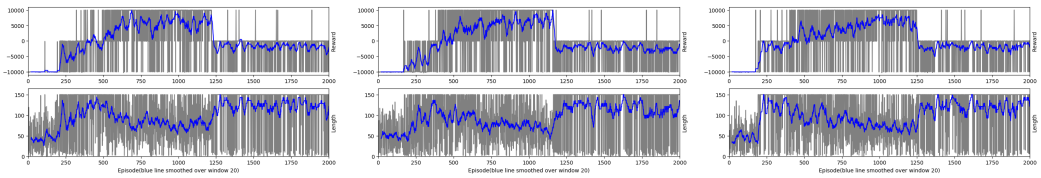
Figura 4.10: Simulaciones realizadas con el agente Experto(Elaboración propia).

y para cada uno de ellos genera máximo 60 reglas, así mismo tambien se muestra el resultado obtenido de la prueba realizada al sistema ya entrenado en el cual el objetivo se alcanza en 1524 episodios lo cual equivale a 76.2 %.

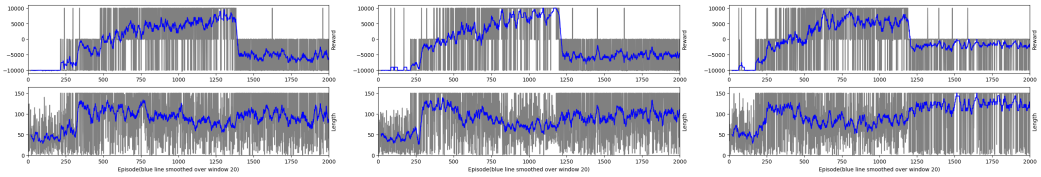
Adicionalmente, se genera un archivo de configuración del agente experto el cual es puede ver en la sección B.2 de los Anexos, este indica el número de secciones en los cuales se seccionará la variable δ_{yaw}, del número de particiones, el número total de reglas, así como el conjunto de reglas, en las cuales cada regla indica la acción a realizar así como un conjunto de condiciones que se deben cumplir para ejecutar dicha regla.

En la figura 4.11 se muestra una comparación de los entrenamientos realizados con diferentes parámetros, en la primera fila se muestran los resultados sin usar segmentación(un grupo), en las siguientes filas 3, 6, 9 y 15 grupos, así mismo en las columnas se muestran 40, 60 y 100 reglas respectivamente por cada segmento, así mismo, en la tabla 4.9 se muestran los porcentajes de episodios finalizados de forma satisfactoria para cada configuración anteriormente mencionada. Con base en las gráficas y los porcentajes mostrados se evidencia que para los casos de segmentación en uno y tres grupos no se generó un conocimiento capaz de resolver la tarea propuesta.

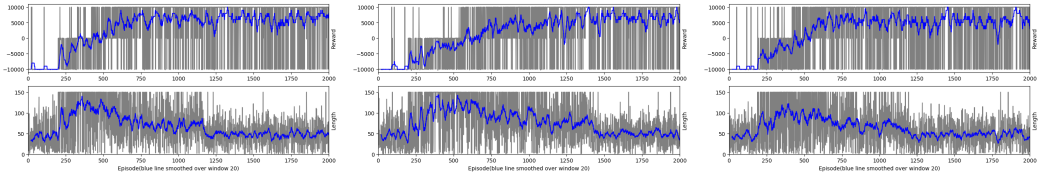
Segmentación en 1 grupo a): 40 reglas ; b) 60 reglas ; c) 100 reglas



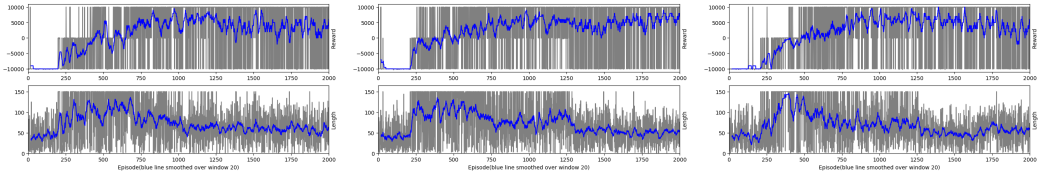
Segmentación en 3 grupos a): 40 reglas ; b) 60 reglas ; c) 100 reglas



Segmentación en 6 grupos a): 40 reglas ; b) 60 reglas ; c) 100 reglas



Segmentación en 9 grupos a): 40 reglas ; b) 60 reglas ; c) 100 reglas



Segmentación en 15 grupos a): 40 reglas ; b) 60 reglas ; c) 100 reglas

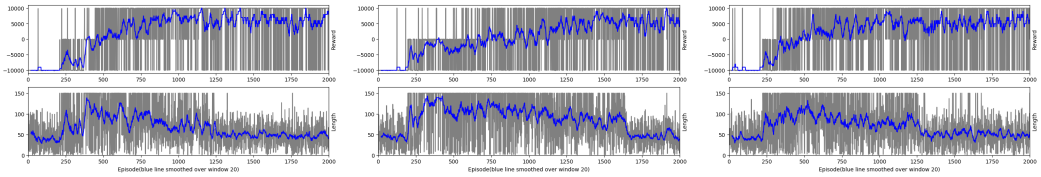


Figura 4.11: Simulaciones realizadas con el agente Experto variando el número de segmentos y el número de reglas por segmento(Elaboración propia).

Segmentos	40 reglas por segmento	60 reglas por segmento	100 reglas por segmento
1	25.35 %	25.3 %	25.35 %
3	25.1 %	25.15 %	25.2 %
6	51.9 %	48.8 %	54.6 %
9	49.85 %	52.2 %	51.4 %
15	55.6 %	38.8 %	52.85 %

Tabla 4.9: Resultados de entrenamiento.

Capítulo 5

CONCLUSIONES Y TRABAJOS FUTUROS

Al determinar los componentes necesarios se determinó la necesidad de generar un preprocesamiento con el fin de reducir el número de variables que serán empleadas por el sistema.

Se observó que al establecer las interacciones entre los componentes que el preprocesamiento de los datos es importante al momento de la generación de las reglas y que dicho procesamiento puede ser diferente entre el componente que realiza la generalización de los datos y el componente que reliza la extracción o generación de las reglas que serán usadas por el sistema experto.

Para obtener un menor número de reglas se evidenció al momento de determinar la metodologia que se requeria un apoyo al momento de realizar la generalización de los datos.

Al evaluar los resultados obtendidos con los diferentes tipos de agentes se observa que el agente experto tiene una efectividad similar al alcazado con las redes neuronales con la gran ventaja de poder auditar y/o modificar el conjunto de reglas del agente, obviamente el ajuste del conjunto de reglas modificará el comportamiento del agente y por ende su efectividad.

En trabajos futuros, sería interesante elaborar un entorno más complejo que permita comportamientos tales como un agente que desplace la ubicación del punto de llegada; múltiples vehículos tratando de alcanzar el punto de llegada; agregar una componente aleatoria al momento de determinar el nuevo estado a partir de la acción seleccionada por el agente; elaborar un entorno tridimensional el cual elevaria

significativamente la complejidad y los tiempos de entrenamiento y simulación. En el caso que se agregen más agentes estos podrían comportarse de manera colaborativa o competitiva.

BIBLIOGRAFÍA

20minutos. (2009). La NASA prueba con éxito el globo atmosférico más grande del mundo. Consultado el 10 de abril de 2016, desde http://www.20minutos.es/noticia/442019/0/nasa/globo/atmosferico/

Alvarez, G. & Flor, O. (2020). Desempeño en métodos de navegación autónoma para robots móviles. *Minerva*, *1*(2), 19-29. https://doi.org/10.47460/minerva.v1i2.8

Andrés, C., Carvajal, V., Alberto, J., Luna, G., Durley, I. & Pardo, T. (2012). Evaluación de las técnicas de planificación de movimientos, Descomposición exacta trapezoidal y Descomposición adaptativa de celdas a través de mallas. *Facultad de ingeniería*, *21*(32), 41-53. https://doi.org/10.19053/01211129.1435

Arbeiza Guerra, F. (2017). *Desarrollo de una plataforma de investigación de deep reinforcement learning*. Cantabria.

Calderón, M. & Camargo Casallas, E. (2020). Training Neural Networks Using Reinforcement Learning To Reactive Path Planning. *Journal of Applied Engineering Science*, *19*(1), 1-9. https://doi.org/10.5937/jaes0-26386

Casallas, E. C., Coronel, C. A., Alberto, L. & Hernández, J. (2020). PROBE-PLATFORM STRATOSPHERIC OF AUTONOMOUS FLIGHT. *International Research Journal of Engineering and Technology*, *07*(08), 3041-3048. https://www.irjet.net/archives/V7/i8/IRJET-V7I8514.pdf

Casallas, E. C., Hernández, L. A. J. & Bello, C. A. B. (2020). On screen display module. *International Journal of Engineering Research and Technology*, *13*(11), 3367-3371. https://doi.org/10.37624/ijert/13.11.2020.3367-3371

Castillo, E., Gutierrez, J. M. & Hadi, A. S. (1997). Sistemas Expertos y Modelos de Redes Probabilísticas, 639.

Chojnacki, E., Plumecocq, W. & Audouin, L. (2019). An expert system based on a Bayesian network for fire safety analysis in nuclear area. *Fire Safety Journal*, *105*(October 2018), 28-40. https://doi.org/10.1016/j.firesaf.2019.02.007

Culbert, C., Riley, G. & Dantes, B. (2015). *CLIPS Reference Manual Volume I. Basic Programming Guide* (Vol. 1).

Gasparetto, A., Boscariol, P., Lanzutti, A. & Vidoni, R. (2015). Path Planning and Trajectory Planning Algorithms: a General Overview. En *Motion and Operation Planning of Robotic Systems* (pp. 3-27). https://doi.org/10.1007/978-3-319-14705-5_18

Gethsiyal Augasta, M. & Kathirvalavakumar, T. (2012). Reverse engineering the neural networks for rule extraction in classification problems. *Neural Processing Letters*, *35*(2), 131-150. https://doi.org/10.1007/s11063-011-9207-8

Goulding, J. R. (1991). A neural network hybrid expert system.

Grupo Artax. (2011). Grupo Artax. Consultado el 1 de abril de 2016, desde http://www.grupoartax.com.ar

Guerrero, J. A. (2013). Cazadores de sondas. Consultado el 2 de abril de 2016, desde http://www.laverdad.es/murcia/rc/20130512/sociedad/cazadores-sondas-201305112023.html

Haines, E. (1994). *Graphics Gems IV (IBM Version)* (P. Heckbert, Ed.). Pittsburgh, Pensilvania, Morgan; kaufmann.

Hasselt, H. V. (2010). Double Q-learning, En *Advances in Neural Informa- tion Processing Systems.*

Hasselt, H. V. (2011). *Insights in Reinforcement Learning: Formal Analysis and Empirical Evaluation of Temporal-Difference Learning Algorithms.*

Hasselt, H. V., Guez, A. & Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning, En *Thirtieth AAAI Conference on Artificial Intelligence.*

Hillcox, C. (2014). 10 ways that a high altitude balloon flight can go wrong. Consultado el 27 de marzo de 2016, desde https://balloonnews.wordpress.com/2014/04/10/10-ways-that-a-high-altitude-balloon-flight-can-go-wrong/

Kartam, N., Flood, I. & Tongthong, T. (1995). Integrating knowledge-based systems and artificial neural networks for engineering. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing*, *9*(1), 13-22. https://doi.org/10.1017/S0890060400002055

Lozano-Pérez, T. (1983). Spatial Planning: A Configuration Space Approach. *IEEE Transactions on Computers*, *C-32*(2), 108-120. https://doi.org/10.1109/TC.1983.1676196

Lozano-Pérez, T. & Wesley, M. A. (1979). An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles. *Communications of the ACM*, *22*(10), 560-570. https://doi.org/10.1145/359156.359164

Mateo Jiménez, F. (2012). *Redes neuronales y preprocesado de variables para modelos y sensores en bioingeniería(Tesis de doctorado)*. Universitat Politècnica de València.

Ministerio de Ciencia Tecnología e Innovación. (2019). DIGITI - SISTEMAS DIGITALES INTELIGENTES. Consultado el 25 de febrero de 2020, desde https://scienti.minciencias.gov.co/gruplac/jsp/visualiza/visualizagr.jsp?nro=00000000004801

Ozden, A., Faghri, A. & Li, M. (2016). Using Knowledge-automation Expert Systems to Enhance the Use and Understanding of Traffic Monitoring Data in State DOTs. *Procedia Engineering*, *145*, 980-986. https://doi.org/10.1016/j.proeng.2016.04.127

Parry, A. C. & Ordoñez, R. (2010). Intelligent Path Planning with Evolutionary Computation. *AIAA Guidance, Navigation, and Control Conference*, (August). https://doi.org/10.2514/6.2010-8039

Perez-Carrasco, R. (2016). What's the fastest way of checking if a point is inside a polygon in python. Consultado el 4 de abril de 2019, desde https://stackoverflow.com/questions/36399381/whats-the-fastest-way-of-checking-if-a-point-is-inside-a-polygon-in-python

Poplin, R., Varadarajan, A. V., Blumer, K., Liu, Y., McConnell, M. V., Corrado, G. S., Peng, L. & Webster, D. R. (2018). Prediction of cardiovascular risk factors from retinal fundus photographs via deep learning. *Nature Biomedical Engineering*, *2*(3), arXiv 1708.09843, 158-164. https://doi.org/10.1038/s41551-018-0195-0

Rich, E. & Knight, K. (1994). *Inteligencia Artificial* (McGraw-Hill, Ed.; Segunda ed). Madrid, Impresos y Revistas. S. A.

Riedmiller, M. (2005). Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *3720 LNAI*, 317-328. https://doi.org/10.1007/11564096_32

Russell, S. & Norving, P. (2016). *Artificial Intelligence A Modern Approach* (Third). Pearson.

Schlatter, T. W. (2009). Atmospheric Composition and Vertical Structure, *6*, 1-54. http://ruc.noaa.gov/AMB%7B%5C_%7DPublications%7B%5C_%7Dbj/2009%20Schlatter%7B%5C_%7DAtmospheric%20Composition%20and%20Vertical%20Structure%7B%5C_%7Deae319MS-1.pdf

Singh, H. & Lone, Y. A. (2020). *Deep Neuro-Fuzzy Systems with Python*. https://doi.org/10.1007/978-1-4842-5361-8

StratoCat. (2014). ¿Que es un globo estratosférico? Consultado el 27 de marzo de 2016, desde http://stratocat.com.ar/artics/globos0.htm

Sunday, D. (2012). Point in polygon inclusion. Consultado el 4 de abril de 2019, desde http://geomalgorithms.com/a03-_inclusion.html

Sutton, R. S. & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (Second edi). MIT Press, Cambridge. http://linkinghub.elsevier.com/retrieve/pii/S1364661399013315

Tan, H. (2017). A brief history and technical review of the expert system research. *IOP Conference Series: Materials Science and Engineering*, *242*(1). https://doi.org/10.1088/1757-899X/242/1/012111

Thrun, S. & Schwartz, A. (1993). Issues in Using Function Approximation for Reinforcement Learning. *Proceedings of the 4th Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1-9.

Vasilev, N., Mincheva, Z. & Nikolov, V. (2020). Decision Tree Extraction using Trained Neural Network, En *Proceedings of the 9th International Conference on Smart Cities and Green ICT Systems*, SCITEPRESS - Science; Technology Publications. https://doi.org/10.5220/0009351801940200

Anexos

Apéndice A

Código fuente

A.1. Estructura del proyecto

En la imagen A.1 se muestra la estructura de carpetas en la cual se organiza el código fuente.

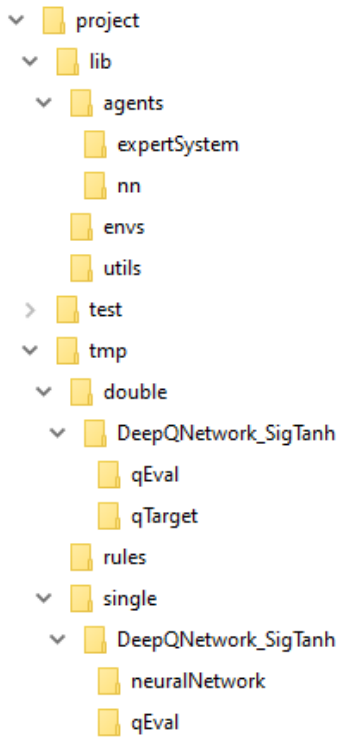


Figura A.1: Organización del código fuente.

A.2. Código fuente

A continuación se muestran los archivos ubicados en la carpeta raíz.

pythonRequiredPackages.txt

```
# To install those packages you must to execute the next command in console
# py -2 -m pip install -r pythonRequiredPackages.txt
numpy == 1.16.1
matplotlib == 2.2.4
tensorflow == 1.13.1
ConfigParser == 3.7.3
scikit-learn == 0.20.4
graphviz == 0.13.2
psycopg2 == 2.7.7
pandas == 0.24.1
etaprogress == 1.1.1
```

RunDoubleDQNAgent.py

```
import numpy as np
import sys

if "../" not in sys.path:
    sys.path.append("../")

from lib.envs.Environment2D import Environment2D as Environment
from lib.agents.DoubleDQNAgent import  DoubleDQNAgent as Agent
from lib.Experiment import Experiment
from lib.EMail import EMail
from lib.Video import Video

# sets the agent operation train/run
# enable training mode
agentOperation = "train"
# enable execution mode
# agentOperation = "run"

# create the environment
env = Environment(xmin = 0, xmax = 10, ymin = 0, ymax = 10, stepLength = 0.1, \
                                                    targetTolerance = 0.1)

# initialize variables
epsilon = 1
epsilonMinimum = 0.01
if(agentOperation == "run"):
    epsilon = epsilonMinimum

# instance the agent to be trained
```

```
agent = Agent(actions = env.actionSpace.actions, \
            inputDimensions = len(env.observation), alpha = 0.0005, \
            gamma = .99, epsilon = epsilon, epsilonDecrement = 0.996, \
            epsilonMinimum = epsilonMinimum, batchSize = 8192, \
            memorySize = 1600000, replaceTarget = 1000)

# load previous trained model if apply
if(agentOperation == "run"):
    print("load data")
    agent.loadModels()

try:
    # creates the experiment
    experiment = Experiment(env, agent, 2000, 150, hidePlots = True, interactive = False)

    # start training
    if(agentOperation == "train"):
        experiment.runAgentLearning()
        agent.saveModels()

    # start executing simulation
    if(agentOperation == "run"):
        experiment.runAgent()

    try:
        # send mail to notify that the main process is already finished
        email = EMail()
        email.sendMail("Finnish :" + agent.__class__.__name__)
    except Exception as e:
        if(experiment.interactive == True):
            print("Error sendig mail")
        else:
            raise

    # if the experiment is interactive creates a video
    if(experiment.interactive == True):
        video = Video(experiment.graph.imagesExperimentPath, 30, 10)
        video.createVideo()
        email.sendMail("Finnish video:" + agent.__class__.__name__ )

except Exception as e:
    # if an error occurrs then send an email to review the failure
    print("===ERROR===")
    print(e)
    print("Oops!", e.__class__, "occurred.")
    email = EMail()
    email.sendMail("ERROR %s \n Oops! %s occurred.\n\n%s"%(agent.__class__.__name__, e.__class__,
     str(e)))
    raise
```

RunExpertAgent.py

```
import sys

if "../" not in sys.path:
    sys.path.append("../")

from lib.envs.Environment2D import Environment2D as Environment
from lib.agents.ExpertAgent import ExpertAgent as Agent
from lib.Experiment import Experiment
from lib.EMail import EMail
from lib.Video import Video

# sets the agent operation train/run
# enable training mode
agentOperation = "train"
# enable execution mode
# agentOperation = "run"

# create the environment
env = Environment(xmin = 0, xmax = 10, ymin = 0, ymax = 10, stepLength = 0.1, \
                                                    targetTolerance = 0.1)

# initialize variables
epsilon = 1
epsilonMinimum = 0.01
if(agentOperation == "run"):
    epsilon = 0
    epsilonMinimum = 0

# instance the agent to be trained
agent = Agent(actions = env.actionSpace.actions, inputDimensions = len(env.observation), \
                alpha = 0.0005, gamma = .99, epsilon = epsilon, \
                epsilonDecrement = 0.996, epsilonMinimum = epsilonMinimum, \
                batchSize = 8192, memorySize = 1600000, \
                groupsNumber = 3600, partitions = 9, rulesByPartition = 60, \
                minimunEpisodesReached = 500)

# load previous trained model if apply
if(agentOperation == "run"):
    print("load data")
    agent.loadModels()

try:
    # creates the experiment
    experiment = Experiment(env, agent, 2000, 150, hidePlots = True, interactive = False)

    # start training
    if(agentOperation == "train"):
        experiment.runAgentLearning()
        # agent.saveModels(experiment.graph.imagesExperimentPath)
        agent.saveModels()

    # start executing simulation
    if(agentOperation == "run"):
        experiment.runAgent()

    try:
        # send mail to notify that the main process is already finished
```

```
58          email = EMail()
59          email.sendMail("Finnish :" + agent.__class__.__name__)
60      except Exception as e:
61          if(experiment.interactive == True):
62              print("Error sendig mail")
63          else:
64              raise
65
66      # if the experiment is interactive creates a video
67      if(experiment.interactive == True):
68          video = Video(experiment.graph.imagesExperimentPath, 30, 10)
69          video.createVideo()
70          email.sendMail("Finnish video:" + agent.__class__.__name__ )
71
72  except Exception as e:
73      # if an error occurrs then send an email to review the failure
74      print("===ERROR===")
75      print(e)
76      print("Oops!", e.__class__, "occurred.")
77      email = EMail()
78      email.sendMail("ERROR %s \n Oops! %s occurred.\n\n%s"%(agent.__class__.__name__, e.__class__,
        str(e)))
79      raise
```

RunRandomAgent.py

```
1   import numpy as np
2   import sys
3
4   if "../" not in sys.path:
5       sys.path.append("../")
6
7   from lib.envs.Environment2D import Environment2D as Environment
8   from lib.agents.RandomAgent import RandomAgent as Agent
9   from lib.Experiment import Experiment
10  from lib.EMail import EMail
11
12  # create the environment
13  env = Environment(xmin = 0, xmax = 10, ymin = 0, ymax = 10, stepLength = 0.1)
14
15  # instance the agent to be used
16  agent = Agent(env.actionSpace.actions)
17
18  try:
19      # creates the experiment
20      experiment = Experiment(env, agent, 2000, 150, hidePlots = True, interactive = False)
21
22      # start executing simulation
23      experiment.runAgent()
24
25      try:
26          # send mail to notify that the main process is already finished
```

```
        email = EMail()
        email.sendMail("Finnish :" + agent.__class__.__name__)
    except Exception as e:
        if(experiment.interactive == True):
            print("Error sendig mail")
        else:
            raise

    if(experiment.interactive == True):
        video = Video(experiment.graph.imagesExperimentPath, 30, 10)
        video.createVideo()
        email.sendMail("Finnish video:" + agent.__class__.__name__ )

except Exception as e:
    # if an error occurrs then send an email to review the failure
    print("===ERROR===")
    print(e)
    print("Oops!", e.__class__, "occurred.")
    email = EMail()
    email.sendMail("ERROR %s \n Oops! %s occurred.\n\n%s"%(agent.__class__.__name__, e.__class__,
     str(e)))
    raise
```

RunSingleDQNAgent.py

```
import numpy as np
import sys

if "../" not in sys.path:
    sys.path.append("../")

from lib.envs.Environment2D import Environment2D as Environment
from lib.agents.SingleDQNAgent import  SingleDQNAgent as Agent
from lib.Experiment import Experiment
from lib.EMail import EMail
from lib.Video import Video

# sets the agent operation train/run
# enable training mode
agentOperation = "train"
# enable execution mode
# agentOperation = "run"

# create the environment
env = Environment(xmin = 0, xmax = 10, ymin = 0, ymax = 10, stepLength = 0.1, \
                                                    targetTolerance = 0.1)
# initialize variables
epsilon = 1
epsilonMinimum = 0.01

if(agentOperation == "run"):
    epsilon = 0
```

```
28      epsilonMinimum = 0
29
30  # instance the agent to be trained
31  agent = Agent(actions = env.actionSpace.actions, \
32              inputDimensions = len(env.observation), alpha = 0.0005, \
33              gamma = .99, epsilon = epsilon, epsilonDecrement = 0.996, \
34              epsilonMinimum = epsilonMinimum, batchSize = 8192, \
35              memorySize = 1600000)
36
37  # load previous trained model if apply
38  if(agentOperation == "run"):
39      print("load data")
40      agent.loadModels()
41
42  try:
43      # creates the experiment
44      experiment = Experiment(env, agent, 2000, 150, hidePlots = True, interactive = False)
45
46      # start training
47      if(agentOperation == "train"):
48          experiment.runAgentLearning()
49          agent.saveModels()
50
51      # start executing simulation
52      if(agentOperation == "run"):
53          experiment.runAgent()
54
55      try:
56          # send mail to notify that the main process is already finished
57          email = EMail()
58          email.sendMail("Finnish :" + agent.__class__.__name__)
59      except Exception as e:
60          if(experiment.interactive == True):
61              print("Error sendig mail")
62          else:
63              raise
64
65      # if the experiment is interactive creates a video
66      if(experiment.interactive == True):
67          video = Video(experiment.graph.imagesExperimentPath, 30, 10)
68          video.createVideo()
69          email.sendMail("Finnish video:" + agent.__class__.__name__ )
70
71  except Exception as e:
72      # if an error occurrs then send an email to review the failure
73      print("===ERROR===")
74      print(e)
75      print("Oops!", e.__class__, "occurred.")
76      email = EMail()
77      email.sendMail("ERROR %s \n Oops! %s occurred.\n\n%s"%(agent.__class__.__name__, e.__class__,
        str(e)))
78      raise
```

A.2.1. lib

EMail.py

```
import smtplib
from email.mime.text import MIMEText

# class used to send notifications
class EMail(object):
    def __init__(self):
        self.smtp_ssl_host = "smtp.gmail.com"
        self.smtp_ssl_port = 465
        self.username = "mvc2.automail@gmail.com"
        self.password = "hr8ZfbmCkH7bUpm"
        self.sender = self.username
        self.targets = ["miltonv.calderonc@gmail.com"]

    # send a notification
    def sendMail(self, message):
        print("...sending mail...")
        msg = MIMEText(message)
        msg["Subject"] = "News"
        msg["To"] = ", ".join(self.targets)
        try:
            server = smtplib.SMTP_SSL(self.smtp_ssl_host, self.smtp_ssl_port)
            server.login(self.username, self.password)
            server.sendmail(self.sender, self.targets, msg.as_string())
            server.quit()

        except Exception as e:
            print("===Email ERROR===")
            print(e)
```

Episode.py

```
class Episode(object):
    def __init__(self, episodeNumber, episodeReward, episodeLength, elapsedTime):
        self.episodeNumber = episodeNumber
        self.episodeReward = episodeReward
        self.episodeLength = episodeLength
        self.elapsedTime = elapsedTime
```

Experiment.py

```
import time
import uuid
import os
from os import system, name
from etaprogress.progress import ProgressBar

from lib.Graph import Graph
from lib.Episode import Episode
from lib.ExperimentData import ExperimentData

# allows the comunication between the environment and the agent
class Experiment(object):
    episodesInformation = []
    # commented to be deleted soon
    simulationGroup = ""
    basePath = ""

    def __init__(self, env, agent, maxNumberEpisodes = 100, maxNumberSteps = 100, \
                                    hidePlots = False, interactive = False):
        self.env = env
        self.agent = agent
        self.hidePlots = hidePlots
        self.interactive = interactive

        self.maxNumberEpisodes = maxNumberEpisodes
        self.maxNumberSteps = maxNumberSteps

        # Variables fot the graphics
        FolderAgent = self.agent.FolderAgent
        self.graph = Graph(env.xmin, env.xmax, env.ymin, env.ymax, \
                    maxNumberEpisodes, maxNumberSteps, FolderAgent, \
                    env.targetTolerance, hidePlots, interactive)
        self.dataLogger = ExperimentData(self.graph.imagesExperimentPath)

        path = os.path.realpath(__file__)
        path, directory_name = os.path.split(path)
        path, directory_name = os.path.split(path)
        self.basePath = path
        self.simulationGroup = str(uuid.uuid4())

    def updateDisplayStep(self, distance = None, reward = None):
        print("...updating step graph...")
        self.graph.updatePlotStep(self.env.trajectory, distance, reward, self.env.
     testPointsLocation)

    def updateDisplayEpisode(self):
        print("...updating episode graph...")
        self.graph.updatePlotEpisode(self.episodesInformation)

    def showGraphExperiment(self):
        print("...showing experiment graph...")
        self.graph.plotExperimentStatistics(self.episodesInformation)

    def runAgent(self):
        print("...running agent...")
        progressBar = ProgressBar(self.maxNumberEpisodes)
        successNumber = 0
```

```
57          for episodeNumber in range(self.maxNumberEpisodes):
58              timeStart = time.time()
59              self.episodeNumber = episodeNumber + 1
60              progressBar.numerator = self.episodeNumber
61
62              # initialize environment
63              state = self.env.reset()
64
65              if self.interactive == True:
66                  self.graph.initializePlotStep(self.env.goal, self.env.death, self.env.warning)
67
68              # used to indicate terminal state
69              done = False
70
71              # used to store the total reward for the episode
72              episodeReward = 0
73
74              stepNumber = 0
75
76              while not done:
77                  stepNumber += 1
78                  self.clearConsole()
79                  print(progressBar)
80                  print("agent type: %s"%(self.agent.getAgentType()))
81                  print("environment mode: %s"%(self.env.environmentMode))
82                  if self.env.environmentMode == "static":
83                      print("environment scenario: "%(self.env.scenario))
84                  print("Max Episode Number: %s Max Step Number: %s"%(self.maxNumberEpisodes, self.
    maxNumberSteps))
85                  print("Episode Number: %s Step Number: %s"%(self.episodeNumber, stepNumber))
86                  print("Success: %s"%successNumber)
87                  # choose action from state
88                  action = self.agent.act(state)
89
90                  # take action, observe reward and next state
91                  nextState, nextReward, done, _ = self.env.step(action)
92
93                  if nextReward >= 9000:
94                      successNumber = successNumber + 1
95
96                  # state <- next state
97                  state = nextState
98
99                  episodeReward += nextReward
100
101                 # Update the graph of the current step
102                 if self.interactive:
103                     self.updateDisplayStep()
104
105                 if(self.maxNumberSteps == stepNumber):
106                     done = True
107
108             timeElapsed = time.time() - timeStart
109             self.episodesInformation.append(Episode(self.episodeNumber, episodeReward, stepNumber,
     timeElapsed))
110             # Update the graph with the data of the current episode
111             if self.interactive:
```

```
112                 self.updateDisplayEpisode()
113 
114         # If is not interactive just show the statistics of the experiment
115         if not self.interactive:
116             self.showGraphExperiment()
117 
118         # store the episode statistics on a csv file
119         self.dataLogger.saveStatistics(self.episodesInformation)
120 
121     def runAgentLearning(self):
122         print("...running agent training...")
123         progressBar = ProgressBar(self.maxNumberEpisodes)
124         successNumber = 0
125         for episodeNumber in range(self.maxNumberEpisodes):
126             timeStart = time.time()
127             self.episodeNumber = episodeNumber + 1
128             progressBar.numerator = self.episodeNumber
129 
130             # initialize environment
131             state = self.env.reset()
132             reward = 0
133 
134             if self.interactive == True:
135                 self.graph.initializePlotStep(self.env.goal, self.env.death, self.env.warning)
136 
137             # used to indicate terminal state
138             done = False
139 
140             # used to store the total reward for the episode
141             episodeReward = 0
142 
143             stepNumber = 0
144 
145             while not done:
146                 stepNumber += 1
147                 self.clearConsole()
148                 print(progressBar)
149                 print("agent type: %s"%(self.agent.getAgentType()))
150                 print("environment mode: %s"%(self.env.environmentMode))
151                 if self.env.environmentMode == "static":
152                     print("environment scenario: "%(self.env.scenario))
153                 print("Max Episode Number: %s Max Step Number: %s"%(self.maxNumberEpisodes, self.
    maxNumberSteps))
154                 print("Episode Number: %s Step Number: %s"%(self.episodeNumber, stepNumber))
155                 print("Success: %s"%successNumber)
156                 # choose action from state
157                 action = self.agent.act(state)
158 
159                 # take action, observe reward and next state
160                 nextState, nextReward, done, _ = self.env.step(action)
161 
162                 # store the transition
163                 self.agent.storeTransition(state, action, nextReward, nextState, done)
164                 if nextReward >= 9000:
165                     successNumber = successNumber + 1
166 
167                 if self.agent.trainingByStep() == True:
```

```
168                     self.agent.learn()
169
170                 # state <- next state
171                 state = nextState
172                 reward = nextReward
173                 episodeReward += nextReward
174
175                 # Update the graph of the current step
176                 if self.interactive:
177                     self.updateDisplayStep(self.env.distance, reward)
178
179                 if(self.maxNumberSteps == stepNumber):
180                     done = True
181
182             # if conditions are fullfiled the episode training is executed
183             if self.agent.trainingByEpisode() == True:
184                 self.agent.learnEpisode()
185
186             timeElapsed = time.time() - timeStart
187             self.episodesInformation.append(Episode(self.episodeNumber, episodeReward, stepNumber,
      timeElapsed))
188             # Update the graph with the data of the current episode
189             if self.interactive:
190                 self.updateDisplayEpisode()
191
192         # If is not interactive just show the statistics of the experiment
193         if not self.interactive:
194             self.showGraphExperiment()
195
196         # store the episode statistics on a csv file
197         self.dataLogger.saveStatistics(self.episodesInformation)
198
199     # clear the python console
200     def clearConsole(self):
201         # for windows
202         if name == "nt":
203             _ = system("cls")
204         # for linux/macos
205         else:
206             _ = system("clear")
```

ExperimentData.py

```
1 import csv
2
3 class ExperimentData(object):
4     def __init__(self, filePath):
5         self.filePath = filePath
6
7     def saveStatistics(self, episodesInformation):
8         print("...saving statistics file...")
9         file = self.filePath + "/experimentStatistics.csv"
```

```
        with open(file, mode="w") as experimentStatistics:
            statisticsWriter = csv.writer(experimentStatistics, delimiter = ";",\
                quotechar = "\"", quoting = csv.QUOTE_MINIMAL)

            statisticsWriter.writerow(["episodeNumber","episodeReward", \
                    "episodeLength", "episodeElapsedTime", "stepAverageElapsedTime"])
            for e in episodesInformation:
                statisticsWriter.writerow([e.episodeNumber, e.episodeReward, \
                    e.episodeLength, e.elapsedTime, e.elapsedTime/e.episodeLength ])
```

Graph.py

```
import os
import datetime
import matplotlib
#line to run matplotlib in just console
matplotlib.use("agg")
from matplotlib import pyplot as plt
from matplotlib.gridspec import GridSpec
import pandas as pd
import math

class Graph(object):
    basePathImages = "Images"
    printDebugMessage = False
    drawTestPoints = True

    def __init__(self, xmin = 0, xmax = 10, ymin = 0, ymax = 0, \
                maxNumberEpisodes = 100, maxNumberSteps = 100, \
                folderAgent = "Nodef", targetTolerance = 0.01, \
                hidePlots = False, interactive = False):
        self.printMessage("...initializing graph...")
        # print "folderAgent: ", folderAgent

        path = os.path.realpath(__file__)
        path, directory_name = os.path.split(path)
        path, directory_name = os.path.split(path)
        self.basePathImages = path + "/" + self.basePathImages

        self.xmin = xmin
        self.ymin = ymin
        self.xmax = xmax
        self.ymax = ymax

        self.maxNumberEpisodes = maxNumberEpisodes
        self.targetTolerance = targetTolerance

        self.hidePlots = hidePlots
        self.interactive = interactive
        # if interactive if false don't draw test points
        if self.drawTestPoints == True and interactive == False:
            self.drawTestPoints = False
```

```
41
42          if self.interactive == True:
43              # self.fig = plt.figure(figsize=(10, 5))
44              my_dpi = 120
45              self.fig = plt.figure(figsize=(1600/my_dpi, 800/my_dpi), dpi=my_dpi)
46              self.gs = GridSpec(2, 2)
47
48              # initialize the step plot
49              self.initializeStepSubplot();
50
51              self.axEpisodeReward = plt.subplot(self.gs[0, 1])
52
53              self.axEpisodeLength = plt.subplot(self.gs[1, 1])
54          else:
55              self.fig = plt.figure(figsize=(10, 5))
56              self.gs = GridSpec(2, 1)
57
58              self.axEpisodeReward = plt.subplot(self.gs[0, 0])
59
60              self.axEpisodeLength = plt.subplot(self.gs[1, 0])
61
62          self.axEpisodeReward.set_xlim(1, maxNumberEpisodes)
63          self.axEpisodeReward.set_ylim(-10000 * 1.1, 10000 * 1.1)
64          self.axEpisodeReward.yaxis.set_label_position("right")
65          self.axEpisodeReward.set_ylabel("Reward")
66
67          self.axEpisodeLength.set_xlim(1, maxNumberEpisodes)
68          self.axEpisodeLength.set_ylim(0, maxNumberSteps * 1.1)
69          self.axEpisodeLength.yaxis.set_label_position("right")
70          self.axEpisodeLength.set_ylabel("Length")
71          self.axEpisodeLength.set_xlabel("Episode")
72
73          self.episodeNumber = 1
74          self.stepNumber = 1
75
76          if self.hidePlots == True or self.hidePlots == False:
77              # Determine the actual time to make a folder to store the images
78              now = datetime.datetime.now().strftime("%Y%m%d%H%M%S")
79
80              if not os.path.isdir(self.basePathImages):
81                  os.makedirs(self.basePathImages)
82
83              self.imagesExperimentPath = self.basePathImages + "/" + folderAgent + \
84                                                                      "_"+ now
85
86              if not os.path.isdir(self.imagesExperimentPath):
87                  os.makedirs(self.imagesExperimentPath)
88
89      def printMessage(self, message):
90          if self.printDebugMessage == None:
91              self.printDebugMessage = False
92          if self.printDebugMessage == True:
93              print message
94
95      def initializeStepSubplot(self):
96          self.axTrajectory = plt.subplot(self.gs[:, 0])
97          # self.axTrajectory.xaxis.set_visible(False)
```

```
98          # self.axTrajectory.yaxis.set_visible(False)
99          xmargin = (self.xmax - self.xmin) / 10
100         ymargin = (self.ymax - self.ymin) / 10
101         self.axTrajectory.set_xlim(self.xmin - xmargin, self.xmax + xmargin)
102         self.axTrajectory.set_ylim(self.ymin - ymargin, self.ymax + ymargin)
103
104     def initializePlotStep(self, target = None, deathZones = [], warningZones = []):
105         if(target == None):
106             target = Point(0, 0, 0)
107         if(deathZones == None):
108             deathZones = []
109         if(warningZones == None):
110             warningZones = []
111
112         self.goal = target
113         self.deathZones = deathZones
114         self.warningZones = warningZones
115
116         # plot world boundaries
117         x=[]
118         y=[]
119         x.append(self.xmin)
120         y.append(self.ymin)
121
122         x.append(self.xmax)
123         y.append(self.ymin)
124
125         x.append(self.xmax)
126         y.append(self.ymax)
127
128         x.append(self.xmin)
129         y.append(self.ymax)
130
131         x.append(self.xmin)
132         y.append(self.ymin)
133
134         self.axTrajectory.plot(x, y, color="black")
135
136         # plot goal point
137         self.axTrajectory.plot(self.goal.x, self.goal.y, marker = "X", color="red")
138
139         goalZone = plt.Circle((self.goal.x, self.goal.y), self.targetTolerance, \
140                                                 color="green", fill = False)
141         self.axTrajectory.add_artist(goalZone)
142
143         #plot death zones
144         for d in self.deathZones:
145             x=[]
146             y=[]
147             for p in d:
148                 x.append(p[0])
149                 y.append(p[1])
150             x.append(d[0][0])
151             y.append(d[0][1])
152             self.axTrajectory.plot(x, y, color="red")
153
154         #plot warning zones
```

```
155         for d in self.warningZones:
156             x=[]
157             y=[]
158             for p in d:
159                 x.append(p[0])
160                 y.append(p[1])
161             x.append(d[0][0])
162             y.append(d[0][1])
163             self.axTrajectory.plot(x, y, color="yellow")
164 
165     def updatePlotStep(self, trajectory, distance = None, reward = None, testPointsLocation = []):
166         self.printMessage("...updating step plot...")
167 
168         #plot trajectory
169         x=[]
170         y=[]
171         for s in trajectory:
172             x.append(s.x)
173             y.append(s.y)
174         self.axTrajectory.plot(x, y, linewidth=0.3, marker=".", color="green")
175 
176         xLabel = "Episode: %05d" % self.episodeNumber +\
177          " Step: %09d" % self.stepNumber
178 
179         if(distance != None):
180             xLabel += "\nDistance: %.3f" % distance
181 
182         if(reward != None):
183             xLabel += "\nReward: %.3f" % reward
184 
185         self.axTrajectory.set_xlabel(xLabel)
186 
187         #draw the position of the test points
188         if self.drawTestPoints == True:
189             for s in testPointsLocation:
190                 xs = []
191                 ys = []
192                 range = s[2]
193                 xs.append(s[0])
194                 ys.append(s[1])
195 
196                 if(range =="large"):
197                     self.axTrajectory.plot(xs, ys, linewidth=0.3, marker=".", color="silver")
198                 elif(range == "middle"):
199                     self.axTrajectory.plot(xs, ys, linewidth=0.3, marker=".", color="yellow")
200                 elif (range == "short"):
201                     self.axTrajectory.plot(xs, ys, linewidth=0.3, marker=".", color="red")
202 
203         if(self.hidePlots == True):
204             self.printMessage("...saving image...")
205             self.saveInteractivePlot()
206 
207         self.stepNumber += 1
208 
209     def updatePlotEpisode(self, episodesInformation):
210         self.printMessage("...updating episode plot...")
211 
```

```
212         self.plotStatistics(episodesInformation)
213         if(self.hidePlots == True):
214             self.printMessage("...saving image...")
215             self.saveInteractivePlot()
216 
217         self.episodeNumber += 1
218         self.stepNumber = 1
219 
220         self.axTrajectory.cla()
221         # Clear the graph for the new episode
222         self.initializeStepSubplot()
223 
224     def plotExperimentStatistics(self, episodesInformation):
225         self.printMessage("...plotting experiment statistics...")
226         self.plotStatistics(episodesInformation)
227         self.saveNonInteractivePlot()
228 
229     def plotStatistics(self, episodesInformation):
230         #plot statistics
231         episodeNumber = []
232         episodeReward = []
233         episodeLength = []
234         for e in episodesInformation:
235             episodeNumber.append(e.episodeNumber)
236             episodeReward.append(e.episodeReward)
237             episodeLength.append(e.episodeLength)
238 
239         size = self.maxNumberEpisodes
240 
241         self.axEpisodeReward.plot(episodeNumber, episodeReward, marker="", color="gray")
242         self.axEpisodeLength.plot(episodeNumber, episodeLength, marker="", color="gray")
243 
244         if(size > 100):
245             sizeGroup = size / 100
246             sizeGroup = math.floor(sizeGroup / 5.0)
247             if(sizeGroup == 0):
248                 sizeGroup = 1
249             size = int(sizeGroup) * 5
250 
251             episodeReward_smoothed = pd.Series(episodeReward).rolling(size, min_periods=size).mean
    ()
252             episodeLength_smoothed = pd.Series(episodeLength).rolling(size, min_periods=size).mean
    ()
253             self.axEpisodeReward.plot(episodeNumber, episodeReward_smoothed, marker="", color="
    blue")
254             self.axEpisodeLength.plot(episodeNumber, episodeLength_smoothed, marker="", color="
    blue")
255 
256             self.axEpisodeLength.set_xlabel("Episode(blue line smoothed over window %d)"% size)
257         else:
258             self.axEpisodeReward.plot(episodeNumber, episodeReward, marker="", color="blue")
259             self.axEpisodeLength.plot(episodeNumber, episodeLength, marker="", color="blue")
260 
261     def saveInteractivePlot(self):
262         imageName = "e%05d" % self.episodeNumber + "_s%09d" % self.stepNumber + ".png"
263         self.fig.savefig(self.imagesExperimentPath + "/" + imageName, bbox_inches="tight")
264 
```

```
265     def saveNonInteractivePlot(self):
266         imageName = "experimentStatistics.png"
267         self.fig.savefig(self.imagesExperimentPath + "/" + imageName, bbox_inches="tight")
```

Settings.py

```
1 # Contains global settings
2 def init():
3     print("...starting global vars...")
4     # determines if debug message will be printed
5     global printDebugMessage
6     printDebugMessage = False
7     # determines level of debug message that will be printed
8     global printDebugLevel
9     printDebugLevel = 0
```

Video.py

```
1 import subprocess
2
3 # create a video from images included in the path
4 class Video(object):
5     # class constructor
6     def __init__(self, path, episodes, maxNumberSteps):
7         print(path)
8         self.path = path
9
10    # create a video from images
11    def createVideo(self):
12        command = "ffmpeg -framerate 30 -pattern_type glob -i '"+self.path+"/*.png' -vf \"pad=ceil
      (iw/2)*2:ceil(ih/2)*2\" -c:v libx264 -r 30 -pix_fmt yuv420p "+self.path+"/video.mp4"
13        output = subprocess.check_output(['bash', '-c', command])
14        print(output)
```

A.2.2. lib/agents

DoubleDQNAgent.py

```
1 import sys
2 import numpy as np
3 import math
```

```
4
5  from lib.agents.DQNAgent import DQNAgent as Agent
6
7  # Is required to select a neural network to be used in the agent
8  # from lib.agents.nn.DeepQNetwork_Relu2 import DeepQNetwork_Relu2 as DeepQNetwork
9  # from lib.agents.nn.DeepQNetwork_Sig2 import DeepQNetwork_Sig2 as DeepQNetwork
10 # from lib.agents.nn.DeepQNetwork_Lin2 import DeepQNetwork_Lin2 as DeepQNetwork
11 # from lib.agents.nn.DeepQNetwork_Tanh2 import DeepQNetwork_Tanh2 as DeepQNetwork
12 # from lib.agents.nn.DeepQNetwork_LinSig import DeepQNetwork_LinSig as DeepQNetwork
13 from lib.agents.nn.DeepQNetwork_SigTanh import DeepQNetwork_SigTanh as DeepQNetwork
14
15 import lib.utils.utilities as utilities
16
17 # this class implements an agent with two NN
18 class DoubleDQNAgent(Agent):
19     networkCloned = 0
20     """Q-Learning agent with function approximation."""
21
22     def __init__(self, actions, inputDimensions, alpha, gamma, epsilon, \
23             epsilonDecrement, epsilonMinimum, batchSize, memorySize, replaceTarget):
24         super(DoubleDQNAgent, self).__init__(actions, inputDimensions, \
25                 alpha, gamma, epsilon, epsilonDecrement, epsilonMinimum, \
26                 batchSize, memorySize)
27         utilities.printMessage("__init__", 1)
28         self.actionSpace = [i for i in range(len(actions))]
29         self.numberActions = len(actions)
30         self.gamma = gamma
31         self.memorySize = memorySize
32         self.memoryCounter = 0
33         self.epsilon = epsilon
34         self.epsilonDecrement = epsilonDecrement
35         self.epsilonMinimum = epsilonMinimum
36         self.batchSize = batchSize
37         self.replaceTarget = replaceTarget
38
39         self.qTarget = DeepQNetwork(alpha, self.numberActions, \
40                 inputDimensions = inputDimensions, \
41                 agentType = "double", neuralNetworkName = "qTarget")
42
43         self.qEval = DeepQNetwork(alpha, self.numberActions, \
44                 inputDimensions = inputDimensions, \
45                 agentType = "double", neuralNetworkName = "qEval")
46
47         self.qTarget.copyNetworkProperties(self.qEval)
48
49         self.stateMemory = np.zeros((self.memorySize, inputDimensions))
50         self.newStateMemory = np.zeros((self.memorySize, inputDimensions))
51
52         self.actionMemory = np.zeros((self.memorySize, self.numberActions),
53                                      dtype = np.int8)
54         self.rewardMemory = np.zeros(self.memorySize)
55         self.terminalMemory = np.zeros(self.memorySize, dtype = np.int8)
56
57         self.FolderAgent = self.__class__.__name__ \
58                             + "_" + self.qEval.__class__.__name__
59
60     # gets an action to apply to the environment
```

```
61      def act(self, state):
62          utilities.printMessage("...getting action...", 1)
63          state = np.array([state], np.float32)
64          rand = np.random.random()
65
66          utilities.printMessage("epsilon: %s"%(self.epsilon), 2)
67          utilities.printMessage("random: %s"%(rand), 2)
68
69          if rand < self.epsilon:
70              utilities.printMessage("...random choice...", 2)
71              action = np.random.choice(self.actionSpace)
72          else:
73              utilities.printMessage("...neural network choice...", 2)
74              utilities.printMessage(state, 2)
75              actions = self.qEval.session.run(self.qEval.QValues,
76                        feed_dict={self.qEval.input: state} )
77              action = np.argmax(actions)
78
79          utilities.printMessage("action: %s"%(action), 2)
80          return action
81
82      # used to train the NN after each step once the batchSize is reached
83      def learn(self):
84          utilities.printMessage("...learning...", 1)
85
86          maxMemory = self.memoryCounter if self.memoryCounter < self.memorySize \
87                                  else self.memorySize
88
89          batch = np.random.choice(maxMemory, self.batchSize)
90
91          stateBatch = self.stateMemory[batch]
92          actionBatch = self.actionMemory[batch]
93          actionValues = np.array(self.actionSpace, dtype = np.int8)
94
95          actionIndices = np.dot(actionBatch, actionValues)
96          rewardBatch = self.rewardMemory[batch]
97          newStateBatch = self.newStateMemory[batch]
98          terminalBatch = self.terminalMemory[batch]
99
100         qNext = self.qTarget.session.run(self.qTarget.QValues,
101                             feed_dict = {self.qTarget.input: newStateBatch})
102
103         qEval = self.qEval.session.run(self.qEval.QValues,
104                                  feed_dict = {self.qEval.input: newStateBatch})
105
106         qPred = self.qEval.session.run(self.qEval.QValues,
107                                  feed_dict = {self.qEval.input: stateBatch})
108
109         maxAction = np.argmax(qEval, axis = 1)
110
111         qTarget = qPred.copy()
112
113         batchIndex = np.arange(self.batchSize)
114
115         qTarget[batchIndex, actionIndices] = rewardBatch + \
116                       self.gamma * np.max(qNext, axis = 1) * terminalBatch
117
```

```
118         _ = self.qEval.session.run(self.qEval.trainOptimizer,
119                         feed_dict = {self.qEval.input: stateBatch,
120                                      self.qEval.actions: actionBatch,
121                                      self.qEval.qTarget: qTarget})
122 
123         if self.memoryCounter % self.replaceTarget == 0:
124             self.qTarget.copyNetworkProperties(self.qEval)
125             self.networkCloned = self.networkCloned + 1
126 
127         utilities.printMessage("network cloned: %s"%(self.networkCloned), 2)
128 
129         self.epsilon = self.epsilon * self.epsilonDecrement if self.epsilon > \
130                                       self.epsilonMinimum else self.epsilonMinimum
131 
132     # load the NN model
133     def loadModels(self):
134         utilities.printMessage("...loading model...", 1)
135         self.qEval.loadCheckpoint()
136         self.qTarget.loadCheckpoint()
137 
138     # save the NN model
139     def saveModels(self):
140         utilities.printMessage( "...saving model...", 1)
141         self.qEval.saveCheckpoint()
142         self.qTarget.saveCheckpoint()
```

DQNAgent.py

```
1  import numpy as np
2 
3  from lib.agents.IAgent import Agent
4  import lib.utils.utilities as utilities
5 
6  # this class implements common methods to DQN and Double DQN agents
7  class DQNAgent(Agent):
8      def __init__(self, actions, inputDimensions, alpha, gamma, epsilon, \
9              epsilonDecrement, epsilonMinimum, batchSize, memorySize):
10         super(DQNAgent, self).__init__(actions)
11 
12     # this method store the information on memory to be processed later
13     def storeTransition(self, state, action, reward, nextState, terminal):
14         utilities.printMessage("...storing transition...", 1)
15         utilities.printMessage("memoryCounter/batchSize: %s/%s"%(self.memoryCounter, self.
       batchSize), 1)
16         index = self.memoryCounter % self.memorySize
17         self.stateMemory[index] = state
18         actions = np.zeros(self.numberActions)
19         actions[action] = 1.0
20         self.actionMemory[index] = actions
21         self.rewardMemory[index] = reward
22         self.newStateMemory[index] = nextState
23         self.terminalMemory[index] = 1 - terminal
```

```
24          self.memoryCounter += 1
```

ExpertAgent.py

```
1  import numpy as np
2  import pandas as pd
3  import io
4  import os
5  import copy
6  import time
7  import math
8  from xml.dom import minidom
9
10 from lib.agents.IAgent import Agent
11 from lib.agents.expertSystem.Condition import Condition
12 from lib.agents.expertSystem.Rule import Rule
13
14 from lib.agents.nn.DeepQNetwork_SigTanh import DeepQNetwork_SigTanh as DeepQNetwork
15
16 from lib.agents.expertSystem.ExpertSystem import ExpertSystem
17 from lib.agents.expertSystem.RulesExtractor import RulesExtractor
18
19 import lib.agents.expertSystem.RuleOperations as ro
20 import lib.utils.utilities as utilities
21
22 # this class contains the functionality of the expert agent
23 class ExpertAgent(Agent):
24     i = 0
25     episodeTrained = 0
26     expertSystem = None
27     rules = None
28     diff = 0
29     steps = 0
30     basePath = None
31     trained = False
32     variables = [
33         "uvectorx"
34         , "uvectory"
35         , "yawvehicle"
36         , "_long0"
37         , "_long05R"
38         , "_long05L"
39         , "_middle15R"
40         , "_middle15L"
41         , "_middle35R"
42         , "_middle35L"
43         , "_short55R"
44         , "_short55L"
45         ]
46
47     groupsNumber = 3600
48     partitions = 9
```

```
49      rulesByPartition = 40
50
51      minimunEpisodesReached = 500
52      episodesReached = None
53      reachedContinuous = 0
54      episodeIncreaseContinuous = 1
55
56      partitionVariables = [
57          "deltayaw"
58          ]
59
60      encodedVariables = [
61          "deltayaw"
62          , "long0"
63          , "long05R"
64          , "long05L"
65          , "middle15R"
66          , "middle15L"
67          , "middle35R"
68          , "middle35L"
69          , "short55R"
70          , "short55L"
71          ]
72
73      # class constructor
74      def __init__(self, actions, inputDimensions, alpha, gamma, epsilon, \
75                  epsilonDecrement, epsilonMinimum, batchSize, memorySize, \
76                  groupsNumber = 3600, partitions = 9, rulesByPartition = 40, \
77                  minimunEpisodesReached = 500):
78          super(ExpertAgent, self).__init__(actions)
79          self.actionSpace = [i for i in range(len(actions))]
80          self.inputDimensions = inputDimensions
81          self.gamma = gamma
82          self.memorySize = memorySize
83          self.memoryCounter = 0
84          self.epsilon = epsilon
85          self.epsilonDecrement = epsilonDecrement
86          self.epsilonMinimum = epsilonMinimum
87          self.batchSize = batchSize
88          self.neuralNetwork = DeepQNetwork(alpha, self.numberActions, \
89                  inputDimensions = self.inputDimensions, \
90                  agentType = "single", neuralNetworkName = "neuralNetwork")
91
92
93          self.groupsNumber = groupsNumber
94          self.partitions = partitions
95          self.rulesByPartition = rulesByPartition
96
97          self.minimunEpisodesReached = minimunEpisodesReached
98          self.episodesReached = 0
99
100         self.rules = []
101
102         path = os.path.realpath(__file__)
103         path, directory_name = os.path.split(path)
104         self.defaultRulesPath = os.path.join(path, "expertSystem", "randomRule.clp")
105         path, directory_name = os.path.split(path)
```

```
106         path, directory_name = os.path.split(path)
107         self.basePath = path
108 
109         self.initializeExpertSystem()
110         self.initializeMemory()
111         self.TrainingMode = "Step/Episode"
112 
113         self.FolderAgent = self.__class__.__name__ \
114                            + "_" + self.neuralNetwork.__class__.__name__
115 
116         self.AgentType = self.__class__.__name__ \
117                            + "_" + self.neuralNetwork.__class__.__name__
118 
119 
120 
121     # initialize the expert system
122     def initializeExpertSystem(self):
123         utilities.printMessage("initializing expert system", 1)
124         self.expertSystem = ExpertSystem(defaultRulesPath = self.defaultRulesPath)
125 
126     # initialize memory space to store information about agent-environment interaction
127     def initializeMemory(self):
128         self.memoryCounter = 0
129         self.stateMemory = np.zeros((self.memorySize, self.inputDimensions))
130         self.newStateMemory = np.zeros((self.memorySize, self.inputDimensions))
131         self.actionMemoryNN = np.zeros((self.memorySize, self.numberActions),
132                                        dtype = np.int8)
133         self.actionMemory = np.zeros((self.memorySize, 1),
134                                        dtype = np.int8)
135         self.rewardMemory = np.zeros(self.memorySize)
136         self.terminalMemory = np.zeros(self.memorySize, dtype = np.int8)
137 
138     # store the transition state caused by some action taken
139     def storeTransition(self, state, action, reward, nextState, terminal):
140         utilities.printMessage("...storing transition...", 1)
141         utilities.printMessage("memoryCounter/batchSize: %s/%s"%(self.memoryCounter, self.
     batchSize), 1)
142         index = self.memoryCounter % self.memorySize
143         self.stateMemory[index] = state
144         actions = np.zeros(self.numberActions)
145         actions[action] = 1.0
146         self.actionMemoryNN[index] = actions
147         self.actionMemory[index] = action
148         self.rewardMemory[index] = reward
149         self.newStateMemory[index] = nextState
150         self.terminalMemory[index] = 1 - terminal
151         self.memoryCounter += 1
152 
153         # the vehicle reach the goal in this episode
154         if reward > 8000:
155             self.episodesReached += 1
156 
157         # if the agent is trained
158         if self.trained == True:
159 
160             if reward > 8000:
161                 self.reachedContinuous += self.episodeIncreaseContinuous
```

```
162                 self.episodeIncreaseContinuous = self.episodeIncreaseContinuous * self.gamma
163                 # if the value decrease the minimum value then the minimum value will be setted
164                 if self.episodeIncreaseContinuous < 0.1:
165                     self.episodeIncreaseContinuous = 0.1
166             elif reward < -8000:
167                 self.reachedContinuous -= 1
168 
169             if self.reachedContinuous < 0:
170                 self.reachedContinuous = 0
171                 self.episodeIncreaseContinuous = 1
172 
173     # encode state to reduce complexity
174     def encondeState(self, state):
175         utilities.printMessage("...encoding state...", 1)
176         encodedState = []
177         yawGoal = np.arctan2(state[1], state[0])
178         yawVehicle = state[2]
179 
180         dYaw = yawGoal - yawVehicle
181         if (dYaw < -math.pi):
182             dYaw += 2 * math.pi
183         elif (dYaw > math.pi):
184             dYaw -= 2 * math.pi
185 
186         #  commented to be deleted soon
187         # if abs(dYaw) > math.pi:
188         #     var = 0/0
189 
190         dYawEncoded = self.encondeYaw(dYaw)
191         # append angle between vehicle orientation and angle to target
192         encodedState.append(dYawEncoded)
193 
194         # append sensors to state
195         for i in range(9):
196             encodedState.append(self.encondeSensor(state[i + 3], self.variables[i + 3]))
197 
198         return encodedState
199 
200     # encode delta yaw requiered to target
201     def encondeYaw(self, yaw):
202         minAngle = -math.pi
203         maxAngle = math.pi
204         delta = (maxAngle - minAngle)/(self.groupsNumber)
205         group = (yaw - minAngle) / delta
206         encoded = int(group)
207         if group.is_integer() == False or encoded == 0:
208             encoded = encoded + 1
209         return encoded
210 
211     # enconde sensors
212     def encondeSensor(self, value, range):
213         atenuator = 0.10
214         free = 0
215         warning = 0
216         denied = 0
217         if "long" in range:
218             denied = -5 * atenuator
```

```
219             warning = -1 * atenuator
220         elif "middle" in range:
221             denied = -7 * atenuator
222             warning = -3 * atenuator
223         elif "short" in range:
224             denied = -9 * atenuator
225             warning = -5 * atenuator
226 
227         if value == free:
228             return 1
229         elif value == warning:
230             return 2
231         elif value == denied:
232             return 3
233 
234     # gets expert system action to apply to the environment
235     def actES(self, state):
236         utilities.printMessage("...expert system choice...", 1)
237         encodedState = self.encondeState(state)
238         deltayaw = "(deltaYaw " + str(encodedState[0]) + ")"
239         sensorValues = "(sensorValues " + str(encodedState[1]) + " " + str(encodedState[2]) +\
240                        " " + str(encodedState[3]) + " " + str(encodedState[4]) +\
241                        " " + str(encodedState[5]) + " " + str(encodedState[6]) +\
242                        " " + str(encodedState[7]) + " " + str(encodedState[8]) + " " + str(
    encodedState[9]) + ")"
243         utilities.printMessage("fact: dyaw: %s"%(deltayaw), 1)
244         utilities.printMessage("fact: sensor: %s"%(sensorValues), 1)
245         factDY = self.expertSystem.assertFact(deltayaw)
246         factSV = self.expertSystem.assertFact(sensorValues)
247         utilities.printMessage("fact: dyaw: %s"%(factDY), 8)
248         utilities.printMessage("fact: sensor: %s"%(factSV), 8)
249         action = self.expertSystem.run()
250         utilities.printMessage("ES action: %s"%(action), 1)
251         return action
252 
253     # gets neural network action to apply to the environment
254     def actNN(self, state):
255         utilities.printMessage("...neural network choice...", 1)
256 
257         state = np.array([state], np.float32)
258 
259         actions = self.neuralNetwork.session.run(self.neuralNetwork.QValues,
260                     feed_dict={self.neuralNetwork.input: state} )
261 
262         action = np.argmax(actions)
263 
264         utilities.printMessage("NN action: %s"%(action), 1)
265         return action
266 
267     # gets an action to apply to the environment
268     def act(self, state):
269         utilities.printMessage("...getting action...", 1)
270         utilities.printMessage("rules: %s"%(len(self.rules)), 2)
271 
272         rand = np.random.random()
273 
274         utilities.printMessage("epsilon: %s"%(self.epsilon), 2)
```

```
275         utilities.printMessage("random: %s"%(rand), 2)
276 
277         if rand < self.epsilon:
278             utilities.printMessage("...random choice...", 2)
279             action = np.random.choice(self.actionSpace)
280         else:
281             actionNN = self.actNN(state)
282             utilities.printMessage("use ES to get actions", 2)
283             utilities.printMessage("number of rules: %s"%len(self.rules), 2)
284             utilities.printMessage("trained: %s"%self.trained, 2)
285             if(self.rules != [] and len(self.rules) > 40) \
286                 or self.trained == True:
287                 # trained property is set to true to keep using ES action
288                 self.trained = True
289 
290                 actionES = self.actES(state)
291                 self.steps += 1
292 
293 
294                 if actionNN != actionES:
295                     self.diff += 1
296 
297                 action = actionES
298 
299                 utilities.printMessage("action NN: %s, action ES: %s, action: %s"%(actionNN,
    actionES, action), 2)
300                 utilities.printMessage("diff/steps: %s/%s"%(self.diff, self.steps), 2)
301             else:
302                 action = actionNN
303         utilities.printMessage("action to apply: %s"%(action), 1)
304         return action
305 
306     # used to train the NN after each step once the batchSize is reached
307     def learn(self):
308         utilities.printMessage("...learning step...", 1)
309         if self.memoryCounter > self.batchSize:
310             maxMemory = self.memoryCounter if self.memoryCounter < self.memorySize \
311                                     else self.memorySize
312 
313             batch = np.random.choice(maxMemory, self.batchSize)
314 
315             stateBatch = self.stateMemory[batch]
316             actionBatch = self.actionMemoryNN[batch]
317             actionValues = np.array(self.actionSpace, dtype = np.int8)
318             actionIndices = np.dot(actionBatch, actionValues)
319             rewardBatch = self.rewardMemory[batch]
320             newStateBatch = self.newStateMemory[batch]
321             terminalBatch = self.terminalMemory[batch]
322 
323             qEval = self.neuralNetwork.session.run(self.neuralNetwork.QValues,
324                                     feed_dict = {self.neuralNetwork.input: stateBatch})
325 
326             qNext = self.neuralNetwork.session.run(self.neuralNetwork.QValues,
327                                 feed_dict = {self.neuralNetwork.input: newStateBatch})
328 
329             qTarget = qEval.copy()
330             batchIndex = np.arange(self.batchSize, dtype = np.int32)
```

```
331
332             qTarget[batchIndex, actionIndices] = rewardBatch + \
333                           self.gamma * np.max(qNext, axis = 1) * terminalBatch
334
335             _ = self.neuralNetwork.session.run(self.neuralNetwork.trainOptimizer,
336                             feed_dict = {self.neuralNetwork.input: stateBatch,
337                                          self.neuralNetwork.actions: actionBatch,
338                                          self.neuralNetwork.qTarget: qTarget})
339
340             self.epsilon = self.epsilon * self.epsilonDecrement if self.epsilon > \
341                                       self.epsilonMinimum else self.epsilonMinimum
342
343     # used to train the ES after reach the minimum episodes were completed succesfully
344     def learnEpisode(self):
345         utilities.printMessage("...learning episode...", 1)
346
347         # the quantity of success episodes was not be reached
348         if self.minimunEpisodesReached > self.episodesReached:
349             return
350
351         # if the system was trained and the counter is positive is not necesary to train again
352         if self.trained == True and self.reachedContinuous > 0:
353             return
354
355         if self.memoryCounter > self.batchSize:
356             maxMemory = self.memoryCounter if self.memoryCounter < self.memorySize \
357                                       else self.memorySize
358
359             encodedStates = map(self.encondeState, self.stateMemory[0:maxMemory])
360
361             dfE = pd.DataFrame(data = encodedStates,    # values
362                 columns = self.encodedVariables)  # 1st row as the column names
363
364             df = pd.DataFrame(data = self.stateMemory[0:maxMemory],    # values
365                 columns = self.variables)  # 1st row as the column names
366
367             for nnColumn in dfE.columns:
368                 column = list(filter(lambda c: c == nnColumn, df.columns))
369                 if(column == []):
370                     df[nnColumn] = dfE[nnColumn]
371
372
373             dr = RulesExtractor()
374
375             rules = dr.getRules(self.neuralNetwork, df, self.variables, self.encodedVariables,
        self.numberActions, self.partitionVariables, self.groupsNumber, self.partitions, self.
        rulesByPartition)
376
377             self.rules = []
378             for rule in rules:
379                 ro.addRule(self.rules, rule)
380
381
382
383             self.expertSystem.clear()
384             self.initializeExpertSystem()
385             for rule in self.rules:
```

```
386                 self.expertSystem.defRule(rule.getClipsRule())
387 
388             self.episodeTrained += 1
389 
390             self.trained = True
391 
392     # load the NN model and ES by loading rules into clips from an xml file
393     def loadModels(self):
394         utilities.printMessage("...loading rules...", 1)
395         self.neuralNetwork.loadCheckpoint()
396         utilities.printMessage("basePath: %s"%(self.basePath), 2)
397         self.rules, self.groupsNumber, self.partitions = self.loadXMLFile(os.path.join(self.
        basePath, "tmp/rules/agentConfig.xml"))
398 
399         self.expertSystem.clear()
400         self.initializeExpertSystem()
401         for rule in self.rules:
402             self.expertSystem.defRule(rule.getClipsRule())
403 
404         self.trained = True
405 
406     # save the NN model and ES rules into an xml file
407     def saveModels(self, path = None):
408         if path == None:
409             path = os.path.join(self.basePath, 'tmp/rules/')
410 
411         utilities.printMessage("...saving nn...", 1)
412         self.neuralNetwork.saveCheckpoint()
413         if(self.rules != None):
414             utilities.printMessage("...saving rules...", 1)
415             self.save2XML(self.rules, self.groupsNumber, self.partitions, os.path.join(path, "
        agentConfigES.xml"))
416 
417     # saves agent configuration into a xml file
418     @staticmethod
419     def save2XML(rules, groupsNumber, partitions, filePath = "agentConfig.xml"):
420         utilities.printMessage("...save rules...", 2)
421         xmlFilePath = filePath
422         xmlRuleList = ""
423         for r in rules:
424             xmlRuleList += r.getXmlDefinition()
425 
426         xmlRules = "<Rules>\n%s\n</Rules>"%xmlRuleList
427         xmlInfo = "<info groups=\"%s\" partitions=\"%s\" rules=\"%s\"></info>"%(groupsNumber,
        partitions, len(rules))
428 
429         xmlText = "<ConfigAgent>\n\
430         %s\n\
431         %s\n\
432         </ConfigAgent>"%(xmlInfo, xmlRules)
433         x = open(xmlFilePath, "w")
434         x.write(xmlText)
435         x.close()
436 
437     # load agent configuration from a xml file
438     @staticmethod
439     def loadXMLFile(filePath):
```

```
440         f = open(filePath, "r")
441         contents = f.read()
442         f.close()
443 
444         return ExpertAgent.loadXMLText(contents)
445 
446     # load agent configuration from xml test
447     @staticmethod
448     def loadXMLText(xmlText):
449         utilities.printMessage("...loading rules...", 2)
450         newRules = []
451         #loads xml file
452         xmlConfiguration = minidom.parseString(xmlText)
453 
454         agentConfiguration = xmlConfiguration.getElementsByTagName("ConfigAgent")[0]
455 
456         information = agentConfiguration.getElementsByTagName("info")[0]
457         groupsNumber = int(information.attributes["groups"].value)
458         partitions = int(information.attributes["partitions"].value)
459 
460         rules = agentConfiguration.getElementsByTagName("Rules")[0]
461 
462         for rule in rules.getElementsByTagName("Rule"):
463 
464             ruleId = rule.attributes["id"].value
465             name = rule.attributes["name"].value
466             action = rule.attributes["action"].value
467             priority = rule.attributes["priority"].value
468 
469             newRule = Rule()
470             newRule.id = ruleId
471             newRule.name = name
472             newRule.action = int(action)
473             newRule.priority = float(priority)
474             newRule.Conditions = []
475 
476             for condition in rule.getElementsByTagName("Condition"):
477 
478                 variable = condition.attributes["variable"].value
479                 operation = condition.attributes["operation"].value
480                 value = condition.attributes["value"].value
481                 fulfilCondition = condition.attributes["fulfilCondition"].value.lower() == "true"
482 
483                 newCondition = Condition(variable, operation, float(value), fulfilCondition)
484                 newRule.conditions.append(newCondition)
485 
486             newRules.append(newRule)
487 
488         return newRules, groupsNumber, partitions
```

IAgent.py

```python
import uuid

import lib.utils.utilities as utilities

# this class defines the minimun methods requiered by an agent
class Agent(object):
    FolderAgent = "noDef"
    TrainingMode = "Step"
    AgentType = "noDef"

    # initialize the class properties
    def __init__(self, actions):
        self.guid = uuid.uuid4()
        self.actions = actions
        self.numberActions = len(actions)
        self.AgentType = self.__class__.__name__
        self.FolderAgent = self.getAgentType()

    # returns the agent type
    def getAgentType(self):
        return self.AgentType

    # this method must be implemented at the specific agent
    def act(self, obs):
        raise NotImplementedError

    # determine if the agent is trained for each episode
    def trainingByEpisode(self):
        return "Episode" in self.TrainingMode

    # determine if the agent is trained for each environment step
    def trainingByStep(self):
        # if the training mode is not by step then is by episode
        utilities.printMessage("training mode: %s"%(self.TrainingMode), 1)
        if ("Step" in self.TrainingMode == False) and ("Episode" in self.TrainingMode == False):
            self.TrainingMode = "Episode"
        return "Step" in self.TrainingMode
```

RandomAgent.py

```python
import numpy as np

from lib.agents.IAgent import Agent

import lib.utils.utilities as utilities

# this class implements an agent who takes random actions
class RandomAgent(Agent):

    def __init__(self, actions):
        super(RandomAgent, self).__init__(actions)
        self.FolderAgent = self.__class__.__name__
```

```
13
14      # gets an action to apply to the environment
15      def act(self, obs):
16          utilities.printMessage("observation: %s"%(obs), 2)
17          action = np.random.randint(0, self.numberActions)
18          return action
```

SingleDQNAgent.py

```
1  import sys
2  import numpy as np
3  import math
4
5  from lib.agents.DQNAgent import DQNAgent as Agent
6  # Is required to select a neural network to be used in the agent
7  # from lib.agents.nn.DeepQNetwork_Relu2 import DeepQNetwork_Relu2 as DeepQNetwork
8  # from lib.agents.nn.DeepQNetwork_Sig2 import DeepQNetwork_Sig2 as DeepQNetwork
9  # from lib.agents.nn.DeepQNetwork_Lin2 import DeepQNetwork_Lin2 as DeepQNetwork
10 # from lib.agents.nn.DeepQNetwork_Tanh2 import DeepQNetwork_Tanh2 as DeepQNetwork
11 # from lib.agents.nn.DeepQNetwork_LinSig import DeepQNetwork_LinSig as DeepQNetwork
12 from lib.agents.nn.DeepQNetwork_SigTanh import DeepQNetwork_SigTanh as DeepQNetwork
13
14 import lib.utils.utilities as utilities
15
16 # this class implements an agent with a single NN
17 class SingleDQNAgent(Agent):
18     """Q-Learning agent with function approximation."""
19
20     def __init__(self, actions, inputDimensions, alpha, gamma, epsilon, \
21             epsilonDecrement, epsilonMinimum, batchSize, memorySize):
22         super(SingleDQNAgent, self).__init__(actions, inputDimensions, \
23                 alpha, gamma, epsilon, epsilonDecrement, epsilonMinimum, \
24                 batchSize, memorySize)
25         utilities.printMessage("__init__", 1)
26         self.actionSpace = [i for i in range(len(actions))]
27         self.numberActions = len(actions)
28         self.gamma = gamma
29         self.memorySize = memorySize
30         self.memoryCounter = 0
31         self.epsilon = epsilon
32         self.epsilonDecrement = epsilonDecrement
33         self.epsilonMinimum = epsilonMinimum
34         self.batchSize = batchSize
35         self.qEval = DeepQNetwork(alpha, self.numberActions, \
36                 inputDimensions = inputDimensions, \
37                 agentType = "single", neuralNetworkName = "qEval")
38         self.stateMemory = np.zeros((self.memorySize, inputDimensions))
39         self.newStateMemory = np.zeros((self.memorySize, inputDimensions))
40         self.actionMemory = np.zeros((self.memorySize, self.numberActions),
41                                      dtype = np.int8)
42         self.rewardMemory = np.zeros(self.memorySize)
43         self.terminalMemory = np.zeros(self.memorySize, dtype = np.int8)
```

```
44
45          self.FolderAgent = self.__class__.__name__ \
46                             + "_" + self.qEval.__class__.__name__
47
48      # gets an action to apply to the environment
49      def act(self, state):
50          utilities.printMessage("...getting action...", 1)
51          state = np.array([state], np.float32)
52          rand = np.random.random()
53
54          utilities.printMessage("epsilon: %s"%(self.epsilon), 2)
55          utilities.printMessage("random: %s"%(rand), 2)
56
57          if rand < self.epsilon:
58              utilities.printMessage("...random choice...", 2)
59              action = np.random.choice(self.actionSpace)
60          else:
61              utilities.printMessage("...neural network choice...", 2)
62              utilities.printMessage(state, 2)
63              actions = self.qEval.session.run(self.qEval.QValues,
64                        feed_dict={self.qEval.input: state} )
65              action = np.argmax(actions)
66
67          utilities.printMessage("action: %s"%(action), 2)
68          return action
69
70      # used to train the NN after each step once the batchSize is reached
71      def learn(self):
72          utilities.printMessage("...learning...", 1)
73          if self.memoryCounter > self.batchSize:
74              maxMemory = self.memoryCounter if self.memoryCounter < self.memorySize \
75                                      else self.memorySize
76
77              batch = np.random.choice(maxMemory, self.batchSize)
78
79              stateBatch = self.stateMemory[batch]
80              actionBatch = self.actionMemory[batch]
81              actionValues = np.array(self.actionSpace, dtype = np.int8)
82              actionIndices = np.dot(actionBatch, actionValues)
83              rewardBatch = self.rewardMemory[batch]
84              newStateBatch = self.newStateMemory[batch]
85              terminalBatch = self.terminalMemory[batch]
86
87              qEval = self.qEval.session.run(self.qEval.QValues,
88                                      feed_dict = {self.qEval.input: stateBatch})
89
90              qNext = self.qEval.session.run(self.qEval.QValues,
91                                  feed_dict = {self.qEval.input: newStateBatch})
92
93              qTarget = qEval.copy()
94              batchIndex = np.arange(self.batchSize, dtype = np.int32)
95
96              qTarget[batchIndex, actionIndices] = rewardBatch + \
97                           self.gamma * np.max(qNext, axis = 1) * terminalBatch
98
99              _ = self.qEval.session.run(self.qEval.trainOptimizer,
100                            feed_dict = {self.qEval.input: stateBatch,
```

```
                                        self.qEval.actions: actionBatch,
                                        self.qEval.qTarget: qTarget})

            self.epsilon = self.epsilon * self.epsilonDecrement if self.epsilon > \
                                    self.epsilonMinimum else self.epsilonMinimum

    # load the NN model
    def loadModels(self):
        utilities.printMessage("...loading model...", 1)
        self.qEval.loadCheckpoint()

    # save the NN model
    def saveModels(self):
        utilities.printMessage( "...saving model...", 1)
        self.qEval.saveCheckpoint()
```

A.2.3. lib/agents/expertSystem

Condition.py

```
class Condition:
    # Condition variable
    variable = None
    # Condition value
    value = None
    # Condition operation
    operation = None
    # Indicates if the condition's operation is fulfilled succesfully or not
    fulfilCondition = None
    # list of compliment operations
    complementOperation = {
        "<=" : ">"
        , ">=" : "<"
        , "<" : ">="
        , ">" : "<="
    }
    xmlOperation = {
          "=" : "eq"
        , "<=" : "le"
        , ">=" : "ge"
        , "<" : "lt"
        , ">" : "gt"
    }
    operationFromXML = {
        "eq" : "="
        , "le" : "<="
        , "ge" : ">="
        , "lt" : "<"
        , "gt" : ">"
    }
    def __init__(self, variable, operation, value, fulfilCondition):
```

```
32          self.variable = variable
33          self.value = value
34          self.operation = operation
35          if self.operationFromXML.has_key(operation):
36              self.operation = self.operationFromXML[operation]
37          self.fulfilCondition = fulfilCondition
38
39      # Set the operation to be performed
40      def setOperation(self, operation):
41          self.operation = operation
42          self.fulfilCondition = True
43
44      # Return the operation to perform
45      def getOperation(self):
46          operation = self.operation
47          if self.fulfilCondition == False:
48              operation = self.getOperationComplement(operation)
49          return operation
50
51      # Return the complement operation
52      def getOperationComplement(self, operation):
53          complement = self.complementOperation[operation]
54          if complement == []:
55              raiseerror("ERROR: complement operation NOT found [" + operation + "]")
56          return complement
57
58      # Return the complement operation
59      def getXmlOperation(self, operation):
60          xmlOperationDefinition = self.xmlOperation[operation]
61          if xmlOperationDefinition == []:
62              raiseerror("ERROR: complement operation NOT found [" + operation + "]")
63          return xmlOperationDefinition
64
65      # print the condition
66      def printCondition(self):
67          condition = ""
68          condition = "NOT "
69          if(self.fulfilCondition == False):
70              print(condition + str(self.variable) + " " + str(self.operation) + " " + str(self.
     value))
71          print(self.getCondition())
72
73      # return the condition
74      def getCondition(self):
75          operation = self.operation
76          if(self.fulfilCondition == False):
77              operation = self.getOperationComplement(self.operation)
78          return str(self.variable) + " " + str(operation) + " " + str(self.value)
79
80      # Return the condition in clips structure
81      def getClipsCondition(self):
82          operation = self.operation
83          if(self.fulfilCondition == False):
84              operation = self.getOperationComplement(self.operation)
85          condition = "(%s ?%s %s)"%(operation, self.variable, str(self.value))
86          return condition
87
```

```
88      # returns the xml definition of the rule
89      def getXmlDefinition(self):
90          xml = "<Condition variable =\"%s\" operation=\"%s\" value=\"%s\" fulfilCondition=\"%s\"/>"
         %(self.variable, self.getXmlOperation(self.operation), self.value, self.fulfilCondition)
91          return xml
```

ExpertSystem.py

```
1  import clips
2  import lib.utils.utilities as utilities
3  import lib.Settings as settings
4
5  # this class wrap clips to centralize its calls
6  class ExpertSystem:
7      defaultRulesPath = None
8      DebugLevel = 9
9
10     # class constructor
11     def __init__(self, defaultRulesPath = "randomRule.clp"):
12         # print default path for ramdom rules
13         # print "path: ", defaultRulesPath
14         self.defaultRulesPath = defaultRulesPath
15         self.reset()
16
17     # reset clips
18     def reset(self):
19         utilities.printMessage("...reset expert system...")
20         utilities.printMessage("loading file: %s"%(self.defaultRulesPath))
21         clips.BatchStar(self.defaultRulesPath)
22         clips.Reset()
23
24     # defines a template
25     def defTemplate(self, templateName, templateSlots, templateComment):
26         utilities.printMessage("templateName : %s"%(templateName))
27         utilities.printMessage("templateSlots: %s"%(templateSlots))
28         utilities.printMessage("templateComments: %s"%(templateComment))
29         clips.BuildTemplate(templateName, templateSlots, templateComment)
30
31     # defines initial facts
32     def defFacts(self, name, facts):
33         utilities.printMessage("deffacts: %s"%(facts))
34         clips.BuildDeffacts(name, facts)
35
36     # Assert a fact into the clips environment
37     def assertFact(self, fact):
38         # clips.PrintFacts()
39         utilities.printMessage("(assert %s)"%(fact))
40         return clips.Assert(fact)
41
42     # Define a new rule into the environment
43     def defRule(self, rule):
44         utilities.printMessage("def rule: %s"%(rule))
```

```
45          clips.Build(rule)
46
47      # Executes the rules to get an action to apply
48      def run(self):
49          utilities.printMessage("====CLIPS FACTS INITIAL====")
50          if settings.printDebugMessage == True and settings.printDebugLevel >= self.DebugLevel:
51              clips.PrintFacts()
52          utilities.printMessage("====CLIPS FACTS INITIAL====")
53          utilities.printMessage("====CLIPS RULES====")
54          if settings.printDebugMessage == True and settings.printDebugLevel >= self.DebugLevel:
55              clips.PrintRules()
56          utilities.printMessage("====CLIPS RULES====")
57
58          clips.Run()
59          t = clips.StdoutStream.Read()
60          # used by tests runing a specific number of steps
61          # t = self.runClipsSteps(1000)
62
63          utilities.printMessage("====CLIPS ANSWER====")
64          utilities.printMessage(t)
65          indexAction = 0
66          label = "action to apply:"
67          if t != None:
68              for line in t.splitlines():
69                  utilities.printMessage(line)
70                  if label in line:
71                      value = line.replace(label,"")
72                      utilities.printMessage("value: %s"%(value))
73                      indexAction = int(value.strip())
74          print("indexAction: %s"%indexAction))
75          utilities.printMessage("====CLIPS ANSWER====")
76          utilities.printMessage("====CLIPS FACTS FINAL====")
77          if settings.printDebugMessage == True and settings.printDebugLevel >= self.DebugLevel:
78              clips.PrintFacts()
79          utilities.printMessage("====CLIPS FACTS FINAL====")
80          return indexAction
81
82      # clear clips
83      def clear(self):
84          clips.Clear()
85
86      # execute all possible clips steps and return the clips output
87      def runClips(self):
88          clips.Run()
89          t = clips.StdoutStream.Read()
90          return t
91
92      # execute a given number of clips steps and return the clips output
93      def runClipsSteps(self, steps):
94          t2 = ""
95          while not clips.Run(steps) < steps:
96              utilities.printMessage("====CLIPS ANSWER====")
97              t = clips.StdoutStream.Read()
98              utilities.printMessage(t)
99              t2 = "%s \n %s"%(t2, t)
100             utilities.printMessage("====CLIPS ANSWER====")
101         return t2
```

IRulesExtractor.py

```
import math
import copy
import numpy as np
import pandas as pd
import sys
import multiprocessing as mp
import time

import lib.agents.expertSystem.RuleOperations as ro

from lib.agents.expertSystem.Rule import Rule
from lib.agents.expertSystem.Condition import Condition

import lib.utils.utilities as utilities

# this class defines the minimun methods requiered by a rule extractor class
class IRulesExtractor(object):

    # defines the method to be used to get the rules
    def getRules(self, neuralNetwork, dataFrame, inputVariables, outputsNumber,
                                    set2Partition, partitions, rulesByPartition):
        raise NotImplementedError

    # Delete the column data(inputs) contained in inputs2Delete
    def deleteInputData(self, dataFrame, inputs2delete = []):
        dataWithoutDeleted = dataFrame.to_numpy()

        if inputs2delete != []:
            for i in inputs2delete:
                dataWithoutDeleted[:, i] = 0

        return dataWithoutDeleted

    # Classify the data and add the actions
    def getClassification(self, neuralNetwork, data):
        dfClassification = copy.deepcopy(data)
        states = dfClassification.to_numpy()

        actions = self.evaluateNetwork(neuralNetwork, states)
        dfClassification["actionNN"] = actions

        return dfClassification

    # Classify the data using the neural network
    def evaluateNetwork(self, neuralNetwork, data):
        utilities.printMessage("...evaluating network...", 2)

        actions2 = neuralNetwork.session.run(neuralNetwork.QValues,
                                        feed_dict = {neuralNetwork.input: data})
        actions2 = np.argmax(actions2, axis = 1)
        actions2 = actions2.reshape((len(data), 1))
        return actions2
```

```
53
54      # gets the accuracy of the test data compared to initial data
55      def getAccuracy(self, initialData, testData):
56          df = pd.DataFrame(data = initialData,    # values
57                            columns = ["initialData"])  # 1st row as the column names
58          df["testData"] = testData
59
60          dataCount = len(df.index)
61          dfOK = df[df["initialData"] == df["testData"]]
62          dataOKCount = len(dfOK.index)
63
64          accuracy = (dataOKCount + 0.0)/dataCount
65
66          return accuracy
67
68      # function to split data into n partitions
69      def splitEncodedData(self, dataFrame, partitionVariable, set2Partition = 3600,
70                                                                  partitions = 3):
71          splittedData = []
72          delta = set2Partition/partitions
73          lowerData = 1
74          for i in range(partitions):
75              upperData = lowerData + delta - 1
76              utilities.printMessage("range: %s, %s"%(lowerData, upperData), 2)
77              partialDataFrame = dataFrame[(lowerData <= dataFrame[partitionVariable])
78                                           & (dataFrame[partitionVariable] <= upperData)]
79              lowerData = upperData + 1
80              splittedData.append(partialDataFrame)
81          return splittedData
82
83      # get the actions for states included in data from the available rules
84      def getActions(self, rules, data, inputVariables):
85          actions = []
86          rulesId = []
87          for state in data:
88              action, ruleId = self.getAction(rules, state, inputVariables)
89              actions.append(action)
90              rulesId.append(ruleId)
91          return actions, rulesId
92
93      # get the action for a state included in data from the available rules
94      def getAction(self, rules, state, inputVariables):
95          for rule in sorted(rules, key = lambda x: x.priority, reverse=True):
96              if self.fulfillRule(rule, state, inputVariables):
97                  return rule.action, rule.id
98
99          return None, None
100
101     # indicates if a state fulfill a rule
102     def fulfillRule(self, rule, state, inputVariables):
103         fulfill = True
104         for condition in rule.conditions:
105             conditionFulfilled = self.fulfillCondition(condition, state, inputVariables)
106             if conditionFulfilled == False:
107                 return False
108             fulfill = fulfill and conditionFulfilled
109         return fulfill
```

```
110
111     # indicates is a state fulfill a condition
112     def fulfillCondition(self, condition, state, inputVariables):
113         conditionFulfilled = False
114         indexVariable = [index for index in range(len(inputVariables))
115                                         if inputVariables[index] == condition.variable]
116         conditionOperation = condition.getOperation()
117         conditionValue = condition.value
118         stateValue = state[indexVariable[0]]
119         if conditionOperation == "<":
120             if stateValue < conditionValue:
121                 conditionFulfilled = True
122         elif conditionOperation == "<=":
123             if stateValue <= conditionValue:
124                 conditionFulfilled = True
125         elif conditionOperation == ">":
126             if stateValue > conditionValue:
127                 conditionFulfilled = True
128         elif conditionOperation == ">=":
129             if stateValue >= conditionValue:
130                 conditionFulfilled = True
131         elif conditionOperation == "=":
132             if stateValue == conditionValue:
133                 conditionFulfilled = True
134         return conditionFulfilled
```

Rule.py

```
1  import uuid
2
3  # this class contains the representation of a rule used in the expert system
4  class Rule:
5      id = None
6      name = None
7      action = None
8      conditions = None
9      priority = None
10
11     # class constructor
12     def __init__(self, name = None):
13         self.id = str(uuid.uuid4())
14         self.conditions = []
15         if name == None:
16             self.name = self.id
17
18     # prints rule conditions
19     def getRuleConditions(self):
20         ruleConditions = ""
21         ruleSeparator = ""
22         for condition in self.conditions:
23             ruleConditions = ruleConditions + ruleSeparator + condition.getCondition()
24             ruleSeparator = " and "
```

```
25          print(ruleConditions)
26
27      # returns conditions on clips structure
28      def getClipsConditions(self):
29          conditions = ""
30          conditionSeparator = ""
31          for condition in self.conditions:
32              conditions = "%s%s%s"%(conditions, conditionSeparator, condition.getClipsCondition())
33              conditionSeparator = " "
34          if len(self.conditions) > 1:
35              conditions = "(test (and %s))"%conditions
36          elif len(self.conditions) == 1:
37              conditions = "(test %s)"%conditions
38          return conditions
39
40      # returns the rule's salience(priority on clips "stack")
41      def getSalience(self):
42          salience = 0
43          if(self.priority < -5000):
44              salience = 10000
45          elif(self.priority > 5000):
46              salience = 5000
47          return salience
48
49      # returns rule in clips structure
50      def getClipsRule(self):
51          clipsCondition = self.getClipsConditions()
52          rule = "(defrule %s \n\
53          (declare (salience %s))\n\
54          ?targetOrientation <- (deltaYaw ?deltayaw)\n\
55          ?actualSensorValues <- (sensorValues ?long0 ?long05R ?long05L ?middle15R ?middle15L ?
    middle35R ?middle35L ?short55R ?short55L)\n\
56          %s\n\
57          =>\n\
58          (retract ?targetOrientation)\n\
59          (retract ?actualSensorValues)\n\
60          (assert (ruleName (time) %s + %s))\n\
61          (assert (action %s))\n\
62          )\n"%(self.id, self.getSalience(), clipsCondition, self.name, self.action, self.action)
63          return rule
64
65      # returns xml definition of the rule
66      def getXmlDefinition(self):
67          xmlConditions = ""
68          conditions = sorted(self.conditions, key = lambda x: x.getOperation())
69          conditions = sorted(conditions, key = lambda x: x.variable)
70          separator = ""
71          for condition in conditions:
72              xmlConditions += separator + condition.getXmlDefinition()
73              separator = "\n"
74          xml = "<Rule id=\"%s\" name=\"%s\" action=\"%s\" priority=\"%s\">\n%s\n</Rule>\n"%(self.id
    , self.name, self.action, self.priority, xmlConditions)
75          return xml
76
77      # returns the variables included
78      def getVariables(self):
79          return {y.variable: y for y in self.conditions}.keys()
```

RuleOperations.py

```
1
2
3  from lib.agents.expertSystem.Condition import Condition
4  from lib.agents.expertSystem.Rule import Rule
5
6  import lib.utils.utilities as utilities
7
8  # determine if a rule is included in another rule
9  def ruleContained(ruleContainer, ruleContained):
10     utilities.printMessage("...rule contained...", 2)
11     utilities.printMessage(". . . . . .", 2)
12     utilities.printMessage("rule %s contains %s"%(ruleContainer.name, ruleContained.name), 2)
13     contained = True
14     for c in ruleContainer.conditions:
15         contained = contained and conditionsContained(c, ruleContained.conditions)
16     utilities.printMessage("...rule contained: %s..."%(contained), 2)
17     return contained
18
19 # determine if a condition is included in a set of conditions
20 def conditionsContained(condition, conditions):
21     utilities.printMessage("...conditions contained...", 2)
22     contained = False
23     operation = condition.getOperation()
24     utilities.printMessage("condition: %s"%(condition.getCondition()), 2)
25
26     if operation == "=":
27         condition2 = filter(lambda x : x.getOperation() == "=" , conditions)
28         if condition2 != None and condition2 != []:
29
30             utilities.printMessage("condition2: %s"%(condition2[0].getCondition()), 2)
31
32             if condition.value == condition2[0].value:
33                 contained = True
34
35     if operation == "<=" or operation == "<":
36         condition2 = filter(lambda x : x.getOperation() == "=" or x.getOperation() == "<=" or x.
       getOperation() == "<" , conditions)
37         if condition2 != None and condition2 != []:
38
39             utilities.printMessage("condition2: %s"%(condition2[0].getCondition()), 2)
40
41             value2 = condition2[0].value
42             operation2 = condition2[0].getOperation()
43
44             if (operation == operation2) and (condition.value == value2):
45                 contained = True
46             if (operation == "<=" or (operation == "<" and operation2 == "<"))\
47                 and (value2 <= condition.value):
48                 contained = True
49             if (operation == "<" and (operation2 == "=" or operation2 == "<="))\
50             and (value2 < condition.value):
51                 contained = True
```

```
52
53      if operation == ">=" or operation == ">":
54          condition2 = filter(lambda x : x.getOperation() == "=" or x.getOperation() == ">=" or x.
        getOperation() == ">" , conditions)
55          if condition2 != None and condition2 != []:
56
57              utilities.printMessage("condition2: %s"%(condition2[0].getCondition()), 2)
58
59              value2 = condition2[0].value
60              operation2 = condition2[0].getOperation()
61
62              if (operation == operation2) and (condition.value == value2):
63                  contained = True
64              if (operation == ">=" or (operation == ">" and operation2 == ">"))\
65                  and (value2 >= condition.value):
66                  contained = True
67              if (operation == ">" and (operation2 == "=" or operation2 == ">="))\
68              and (value2 > condition.value):
69                  contained = True
70
71      utilities.printMessage(contained)
72      return contained
73
74  # simplify rule's conditions
75  def simplifyConditions(rule):
76      utilities.printMessage("...simplifing rule conditions...", 2)
77      utilities.printMessage("rule name: %s"%(rule.name), 2)
78      newRule = rule
79      initialConditions = rule.conditions
80      newRule.conditions = []
81
82      for condition in initialConditions:
83          if newRule != None:
84              newRule = simplifyRuleAddCondition(newRule, condition)
85
86      return newRule
87
88  # simplify rules condition when a new condition is added
89  def simplifyRuleAddCondition(rule, newCondition):
90      utilities.printMessage("...adding condition...", 2)
91
92      addCondition = True
93      deleteRule = False
94
95      ruleConditions = filter(lambda x: x.variable == newCondition.variable, rule.conditions)
96
97      utilities.printMessage("conditionsSameVariable: %s"%(len(ruleConditions)), 2)
98      if len(ruleConditions) == 0:
99          addCondition = True
100
101     for ruleCondition in sorted(ruleConditions, key=lambda x: x.getOperation()):
102
103         utilities.printMessage("rule condition: %s"%(ruleCondition.getXmlDefinition()), 2)
104         utilities.printMessage("new condition:  %s"%(newCondition.getXmlDefinition()), 2)
105
106         operation = ruleCondition.getOperation()
107         operation2 = newCondition.getOperation()
```

```
108         value1 = ruleCondition.value
109         value2 = newCondition.value
110         utilities.printMessage("operation: %s operation2: %s"%(operation, operation2), 2)
111         if operation == "=" and operation2 == "=":
112             utilities.printMessage("caso 1", 2)
113             if value1 != value2:
114                 deleteRule = True
115         elif (operation == "<" or operation == "<=") and (operation2 == "<" or operation2 == "<=")
    :
116             utilities.printMessage("caso 5", 2)
117             addCondition = False
118             if value1 == value2:
119                 ruleCondition.setOperation("<")
120                 if operation == "<=" and operation2 == "<=":
121                     ruleCondition.setOperation("<=")
122             elif value1 < value2:#this condition can be removed, but is just for map all
    posibilities
123                 ruleCondition.value = value1
124                 ruleCondition.setOperation(operation)
125             elif value1 > value2:
126                 ruleCondition.value = value2
127                 ruleCondition.setOperation(operation2)
128         elif (operation == ">" or operation == ">=") and (operation2 == ">" or operation2 == ">=")
    :
129             addCondition = False
130             utilities.printMessage("caso 9", 2)
131             if value1 == value2:
132                 ruleCondition.setOperation(">")
133                 if operation == ">=" and operation2 == ">=":
134                     ruleCondition.setOperation(">=")
135             elif value1 < value2:
136                 utilities.printMessage("setting value: %s"%(value2), 2)
137                 ruleCondition.value = value2
138                 ruleCondition.setOperation(operation2)
139             elif value1 > value2:#this condition can be removed, but is just for map all
    posibilities
140                 utilities.printMessage("setting value: %s"%(value1), 2)
141                 ruleCondition.value = value1
142                 ruleCondition.setOperation(operation)
143             utilities.printMessage("rule condition (9): %s"%(ruleCondition.getXmlDefinition()), 2)
144         # operations with = against <= or <
145         elif ((operation == "=") and (operation2 == "<" or operation2 == "<=")) \
146             or ((operation2 == "=") and (operation == "<" or operation == "<=")):
147             addCondition = False
148             utilities.printMessage("caso 2 4", 2)
149             if (operation == "="):
150                 utilities.printMessage("caso 4", 2)
151                 valueA = value2
152                 valueB = value1
153                 operationA = operation2
154                 operationB = operation
155             else:
156                 utilities.printMessage("caso 2", 2)
157                 valueA = value1
158                 valueB = value2
159                 operationA = operation
160                 operationB = operation2
```

```
161             if (operationA == "<="):
162                 if (valueB <= valueA):
163                     ruleCondition.setOperation("=")
164                     ruleCondition.value = valueB
165                 else:
166                     deleteRule = True
167             else:
168                 if (valueB < valueA):
169                     ruleCondition.setOperation("=")
170                     ruleCondition.value = valueB
171                 else:
172                     deleteRule = True
173         # operations with = against >= or >
174         elif ((operation == "=") and (operation2 == ">" or operation2 == ">=")) \
175             or ((operation2 == "=") and (operation == ">" or operation == ">=")):
176             addCondition = False
177             utilities.printMessage("caso 3 7", 2)
178             if (operation == "="):
179                 utilities.printMessage("caso 7", 2)
180                 valueA = value2
181                 valueB = value1
182                 operationA = operation2
183                 operationB = operation
184             else:
185                 utilities.printMessage("caso 3", 2)
186                 valueA = value1
187                 valueB = value2
188                 operationA = operation
189                 operationB = operation2
190             if (operationA == ">="):
191                 if (valueB >= valueA):
192                     ruleCondition.setOperation("=")
193                     ruleCondition.value = valueB
194                 else:
195                     deleteRule = True
196             else:
197                 if (valueB > valueA):
198                     ruleCondition.setOperation("=")
199                     ruleCondition.value = valueB
200                 else:
201                     deleteRule = True
202         # operations with < or <=  against > or >=
203         elif ((operation == "<" or operation == "<=") and (operation2 == ">" or operation2 == ">="
    )) \
204             or ((operation == ">" or operation == ">=") and (operation2 == "<" or operation2 == "
    <=")):
205 
206             utilities.printMessage("caso 6 8", 2)
207             if (operation == ">=" or operation == ">"):
208                 utilities.printMessage("caso 6", 2)
209                 valueA = value1
210                 valueB = value2
211                 operationA = operation
212                 operationB = operation2
213             else:
214                 utilities.printMessage("caso 8", 2)
215                 valueA = value2
```

```
216                 valueB = value1
217                 operationA = operation2
218                 operationB = operation
219 
220             if valueA == valueB:
221                 if operationA == ">=" and operationB == "<=":
222                     addCondition = False
223                     ruleCondition.setOperation("=")
224                 else:
225                     deleteRule = True
226             elif valueA < valueB:
227                 addCondition = addCondition and True
228             elif valueA > valueB:
229                 deleteRule = True
230 
231     if addCondition == True:
232         rule.conditions.append(newCondition)
233 
234     if deleteRule == True:
235         rule = None
236 
237     if rule == None:
238         utilities.printMessage("rule deleted", 2)
239     else:
240         utilities.printMessage(rule.getXmlDefinition(), 2)
241     return rule
242 
243 # add a rule to a set of rules
244 def addRule(rules, rule):
245     addRule = False
246     if rules == []:
247         addRule = True
248 
249     utilities.printMessage("adding rule: %s"%(rule.id), 2)
250     for r in rules:
251         if r.action == rule.action:
252             if ruleContained(r, rule) == True:
253                 utilities.printMessage("rule contained", 2)
254                 return
255             if ruleContained(rule, r) == True:
256                 utilities.printMessage("rule replaced", 2)
257                 r = rule
258                 return
259         else:
260             if ruleContained(r, rule) == True:
261                 if rule.priority < r.priority + 100:
262                     rule.priority = r.priority + 100
263             if ruleContained(rule, r) == True:
264                 if r.priority < rule.priority + 100:
265                     r.priority = rule.priority + 100
266             addRule = True
267 
268     if addRule == True:
269         rules.append(rule)
270 
271 # print the xml definition from a set of rules
272 def printRules(rules):
```

```
273     for rule in rules:
274         print(rule.getXmlDefinition())
```

RulesExtractor.py

```
1  import math
2  import copy
3  import numpy as np
4  import pandas as pd
5  import sys
6  import multiprocessing as mp
7  import time
8
9  from lib.agents.expertSystem.IRulesExtractor import IRulesExtractor
10
11 from lib.agents.expertSystem.Rule import Rule
12 from lib.agents.expertSystem.Condition import Condition
13
14 import lib.utils.utilities as utilities
15
16 class RulesExtractor(IRulesExtractor):
17
18     # Extract rules from classified data
19     def getRules(self, neuralNetwork, dataFrame, nnVariables, esVariables, outputsNumber,
20                  partitionVariables, set2Partition, partitions, rulesByPartition):
21         rules = []
22
23         utilities.printMessage("totalData: %s"%(len(dataFrame.index)), 3)
24
25         nnDataFrame = dataFrame[nnVariables]
26         # takes just the needed data
27         dataFrame = dataFrame[esVariables]
28         utilities.printMessage("esVariables: %s"%(esVariables), 3)
29         # delete duplicates
30         utilities.printMessage("removing duplicates")
31         dataFrame = dataFrame.drop_duplicates(keep="first")
32         # print records to process
33         utilities.printMessage("totalData: %s"%(len(dataFrame.index)), 3)
34
35         dfClassification = self.getClassification(neuralNetwork, nnDataFrame)
36
37         dataFrame["actionNN"] = dfClassification["actionNN"]
38
39         dfProcess = []
40         # optional split data
41         partitionVariable = partitionVariables[0]
42         dfProcess = self.splitEncodedData(dataFrame, partitionVariable, set2Partition
43                                                                       , partitions)
44
45         requiredProperties = copy.deepcopy(esVariables)
46         requiredProperties.append("actionNN")
47         utilities.printMessage("requiredProperties: %s"%(requiredProperties), 3)
```

```
48
49          for dfProcessing in dfProcess:
50              # use just the required columns
51              dfProcessing = dfProcessing[requiredProperties]
52
53              # first grouping properties
54              grouping1Properties = copy.deepcopy(esVariables)
55              # removing partitionVariables
56              for pv in partitionVariables:
57                  grouping1Properties.remove(pv)
58              # adding action property
59              grouping1Properties.append("actionNN")
60
61              # group data and calculates the size of the group, min, max and mean of deltayaw
62              groupedData = dfProcessing.groupby(grouping1Properties).agg(
63                      [np.size, np.min, np.max, np.mean]
64              )
65
66              # rename columns to convert into string
67              groupedData.columns =  ["_".join(x) for x in groupedData.columns.ravel()]
68              groupedData = groupedData.reset_index()
69              groupedData = groupedData.to_numpy(dtype="int32")
70
71              # define properties to create a new dataFrame with grouped data
72              propertiesGroup1 = copy.deepcopy(grouping1Properties)
73              propertiesGroup1.append("size")
74              propertiesGroup1.append("min")
75              propertiesGroup1.append("max")
76              propertiesGroup1.append("mean")
77
78              # creates a dataFrame from grouped data using columns listed on propertiesGroup1
79              groupedDataFrame = pd.DataFrame(data = groupedData,    # values
80                  columns = propertiesGroup1)  # 1st row as the column names
81
82              # use just the required columns
83              varItems = ["actionNN", "size"]
84              groupedDataFrameTmp = groupedDataFrame[varItems]
85
86              # second grouping properties
87              grouping2Properties = ["actionNN"]
88
89              # grouping over first group
90              # then calculate size and group each one divided over the total sum
91              groupedDataC = groupedDataFrameTmp.groupby(grouping2Properties).agg([np.size,
92                                          np.sum]).apply(lambda x: x / float(x.sum()))
93
94              # renaming columns
95              groupedDataC.columns =  ["_".join(x) for x in groupedDataC.columns.ravel()]
96
97              # average value
98              groupedDataC["avg"] = groupedDataC[["size_size", "size_sum"]].mean(axis=1)
99
100             # defines default action
101             defaultAction = groupedDataC["avg"].idxmax()
102
103             # generate default action(just deltayaw with all range)
104             rule = Rule()
```

```
105             rule.action = defaultAction
106             rule.priority = -1000
107 
108             for pVar in partitionVariables:
109                 lowerValue = groupedDataFrame["min"].min()
110                 upperValue = groupedDataFrame["max"].max()
111 
112                 conditionL = Condition(pVar, ">=", lowerValue, True)
113                 rule.conditions.append(conditionL)
114                 conditionU = Condition(pVar, "<=", upperValue, True)
115                 rule.conditions.append(conditionU)
116 
117             rules.append(rule)
118 
119             # generate another rules
120             # order descendent the first group
121             dfGenerateRules = groupedDataFrame[groupedDataFrame["actionNN"] != defaultAction].
    sort_values(by=["size"], ascending=False)
122 
123             numberOfRules = rulesByPartition
124             ruleCount = 1
125             for index, row in dfGenerateRules.iterrows():
126                 # if the number of rules is complete
127                 if ruleCount >= numberOfRules:
128                     break
129 
130                 # create and add rule
131                 rule = Rule()
132                 rule.action = row["actionNN"]
133                 rule.priority = 0
134 
135                 # iterates over all ES variables
136                 for i in esVariables:
137                     lowerValue = 0
138                     upperValue = 0
139 
140                     if i in partitionVariables:
141                         lowerValue = row["min"]
142                         upperValue = row["max"]
143                     else:
144                         lowerValue = row[i]
145                         upperValue = row[i]
146 
147                     conditionL = Condition(i, ">=", lowerValue, True)
148                     rule.conditions.append(conditionL)
149                     conditionU = Condition(i, "<=", upperValue, True)
150                     rule.conditions.append(conditionU)
151 
152                 # add the new rule
153                 rules.append(rule)
154                 ruleCount = ruleCount + 1
155 
156         # return the rules
157         return rules
```

A.2.4. lib/agents/nn

DeepQNetwork_Lin2.py

```
from lib.agents.nn.IDeepQNetwork import IDeepQNetwork
import tensorflow as tf
import numpy as np

class DeepQNetwork_Lin2(IDeepQNetwork):
    def __init__(self, learningRate, numberActions, inputDimensions, \
                    agentType, neuralNetworkName):
        print("__init__")

        self.nodesLayer1 = 256
        self.nodesLayer2 = 256

        super(DeepQNetwork_Lin2, self).__init__(learningRate, \
                    numberActions, inputDimensions, \
                    agentType, neuralNetworkName)


    def buildNetwork(self):
        print("...build network...")
        with tf.variable_scope(self.neuralNetworkName):
            print("...building placeholders...")
            self.input = tf.placeholder(tf.float32,
                                        shape = [None, self.inputDimensions],
                                        name = "inputs")
            self.actions = tf.placeholder(tf.float32,
                                          shape = [None, self.numberActions],
                                          name = "actionTaken")
            self.qTarget = tf.placeholder(tf.float32,
                                           shape = [None, self.numberActions],
                                           name = "qValue")

            print("...building layers...")
            flat = tf.layers.flatten(self.input)
            dense1 = tf.layers.dense(flat, units = self.nodesLayer1,
                                     activation = None,)
            dense2 = tf.layers.dense(dense1, units = self.nodesLayer2,
                                     activation = None,)
            self.QValues = tf.layers.dense(dense2, units = self.numberActions,)

            self.loss = tf.reduce_mean(tf.square(self.QValues - self.qTarget))
            self.trainOptimizer = tf.train.AdamOptimizer(self.learningRate).minimize(self.loss)
```

DeepQNetwork_LinSig.py

```python
from lib.agents.nn.IDeepQNetwork import IDeepQNetwork
import tensorflow as tf
import numpy as np

class DeepQNetwork_LinSig(IDeepQNetwork):
    def __init__(self, learningRate, numberActions, inputDimensions, \
                    agentType, neuralNetworkName):
        print("__init__")

        self.nodesLayer1 = 256
        self.nodesLayer2 = 256

        super(DeepQNetwork_LinSig, self).__init__(learningRate, \
                    numberActions, inputDimensions, \
                    agentType, neuralNetworkName)


    def buildNetwork(self):
        print("...build network...")
        with tf.variable_scope(self.neuralNetworkName):
            print("...building placeholders...")
            self.input = tf.placeholder(tf.float32,
                                        shape = [None, self.inputDimensions],
                                        name = "inputs")
            self.actions = tf.placeholder(tf.float32,
                                          shape = [None, self.numberActions],
                                          name = "actionTaken")
            self.qTarget = tf.placeholder(tf.float32,
                                          shape = [None, self.numberActions],
                                          name = "qValue")

            print("...building layers...")
            flat = tf.layers.flatten(self.input)
            dense1 = tf.layers.dense(flat, units = self.nodesLayer1,
                                     activation = None,)
            dense2 = tf.layers.dense(dense1, units = self.nodesLayer2,
                                     activation = tf.nn.sigmoid,)
            self.QValues = tf.layers.dense(dense2, units = self.numberActions,)

            self.loss = tf.reduce_mean(tf.square(self.QValues - self.qTarget))
            self.trainOptimizer = tf.train.AdamOptimizer(self.learningRate).minimize(self.loss)
```

DeepQNetwork_Relu2.py

```python
from lib.agents.nn.IDeepQNetwork import IDeepQNetwork
import tensorflow as tf
import numpy as np

class DeepQNetwork_Relu2(IDeepQNetwork):
    def __init__(self, learningRate, numberActions, inputDimensions, \
                    agentType, neuralNetworkName):
        print("__init__")
```

```
9
10          self.nodesLayer1 = 256
11          self.nodesLayer2 = 256
12
13          super(DeepQNetwork_Relu2, self).__init__(learningRate, \
14                      numberActions, inputDimensions, \
15                      agentType, neuralNetworkName)
16
17
18      def buildNetwork(self):
19          print("...build network...")
20          with tf.variable_scope(self.neuralNetworkName):
21              print("...building placeholders...")
22              self.input = tf.placeholder(tf.float32,
23                                          shape = [None, self.inputDimensions],
24                                          name = "inputs")
25              self.actions = tf.placeholder(tf.float32,
26                                            shape = [None, self.numberActions],
27                                            name = "actionTaken")
28              self.qTarget = tf.placeholder(tf.float32,
29                                            shape = [None, self.numberActions],
30                                            name = "qValue")
31
32              print("...building layers...")
33              flat = tf.layers.flatten(self.input)
34              dense1 = tf.layers.dense(flat, units = self.nodesLayer1,
35                                       activation = tf.nn.relu,)
36              dense2 = tf.layers.dense(dense1, units = self.nodesLayer2,
37                                       activation = tf.nn.relu,)
38              self.QValues = tf.layers.dense(dense2, units = self.numberActions,)
39
40              self.loss = tf.reduce_mean(tf.square(self.QValues - self.qTarget))
41              self.trainOptimizer = tf.train.AdamOptimizer(self.learningRate).minimize(self.loss)
```

DeepQNetwork_Sig2.py

```
1  from lib.agents.nn.IDeepQNetwork import IDeepQNetwork
2  import tensorflow as tf
3  import numpy as np
4
5  class DeepQNetwork_Sig2(IDeepQNetwork):
6      def __init__(self, learningRate, numberActions, inputDimensions, \
7                      agentType, neuralNetworkName):
8          print("__init__")
9
10         self.nodesLayer1 = 256
11         self.nodesLayer2 = 256
12
13         super(DeepQNetwork_Sig2, self).__init__(learningRate, \
14                     numberActions, inputDimensions, \
15                     agentType, neuralNetworkName)
16
```

```
17
18      def buildNetwork(self):
19          print("...build network...")
20          with tf.variable_scope(self.neuralNetworkName):
21              print("...building placeholders...")
22              self.input = tf.placeholder(tf.float32,
23                                          shape = [None, self.inputDimensions],
24                                          name = "inputs")
25              self.actions = tf.placeholder(tf.float32,
26                                            shape = [None, self.numberActions],
27                                            name = "actionTaken")
28              self.qTarget = tf.placeholder(tf.float32,
29                                            shape = [None, self.numberActions],
30                                            name = "qValue")
31
32              print("...building layers...")
33              flat = tf.layers.flatten(self.input)
34              dense1 = tf.layers.dense(flat, units = self.nodesLayer1,
35                                       activation = tf.nn.sigmoid,)
36              dense2 = tf.layers.dense(dense1, units = self.nodesLayer2,
37                                       activation = tf.nn.sigmoid,)
38              self.QValues = tf.layers.dense(dense2, units = self.numberActions,)
39
40              self.loss = tf.reduce_mean(tf.square(self.QValues - self.qTarget))
41              self.trainOptimizer = tf.train.AdamOptimizer(self.learningRate).minimize(self.loss)
```

DeepQNetwork_SigTanh.py

```
1  from lib.agents.nn.IDeepQNetwork import IDeepQNetwork
2  import tensorflow as tf
3  import numpy as np
4
5  class DeepQNetwork_SigTanh(IDeepQNetwork):
6      def __init__(self, learningRate, numberActions, inputDimensions, \
7                   agentType, neuralNetworkName):
8          print("__init__")
9
10         self.nodesLayer1 = 256
11         self.nodesLayer2 = 256
12
13         super(DeepQNetwork_SigTanh, self).__init__(learningRate, \
14                     numberActions, inputDimensions, \
15                     agentType, neuralNetworkName)
16
17
18     def buildNetwork(self):
19         print("...build network...")
20         with tf.variable_scope(self.neuralNetworkName):
21             print("...building placeholders...")
22             self.input = tf.placeholder(tf.float32,
23                                         shape = [None, self.inputDimensions],
24                                         name = "inputs")
```

```
            self.actions = tf.placeholder(tf.float32,
                                          shape = [None, self.numberActions],
                                          name = "actionTaken")
            self.qTarget = tf.placeholder(tf.float32,
                                           shape = [None, self.numberActions],
                                           name = "qValue")

            print("...building layers...")
            flat = tf.layers.flatten(self.input)
            dense1 = tf.layers.dense(flat, units = self.nodesLayer1,
                                     activation = tf.nn.sigmoid,)
            dense2 = tf.layers.dense(dense1, units = self.nodesLayer2,
                                     activation = tf.nn.tanh,)
            self.QValues = tf.layers.dense(dense2, units = self.numberActions,)

            self.loss = tf.reduce_mean(tf.square(self.QValues - self.qTarget))
            self.trainOptimizer = tf.train.AdamOptimizer(self.learningRate).minimize(self.loss)
```

DeepQNetwork_Tanh2.py

```
from lib.agents.nn.IDeepQNetwork import IDeepQNetwork
import tensorflow as tf
import numpy as np

class DeepQNetwork_Tanh2(IDeepQNetwork):
    def __init__(self, learningRate, numberActions, inputDimensions, \
                       agentType, neuralNetworkName):
        print("__init__")

        self.nodesLayer1 = 256
        self.nodesLayer2 = 256

        super(DeepQNetwork_Tanh2, self).__init__(learningRate, \
                    numberActions, inputDimensions, \
                    agentType, neuralNetworkName)


    def buildNetwork(self):
        print("...build network...")
        with tf.variable_scope(self.neuralNetworkName):
            print("...building placeholders...")
            self.input = tf.placeholder(tf.float32,
                                        shape = [None, self.inputDimensions],
                                        name = "inputs")
            self.actions = tf.placeholder(tf.float32,
                                          shape = [None, self.numberActions],
                                          name = "actionTaken")
            self.qTarget = tf.placeholder(tf.float32,
                                           shape = [None, self.numberActions],
                                           name = "qValue")

            print("...building layers...")
```

```
            flat = tf.layers.flatten(self.input)
            dense1 = tf.layers.dense(flat, units = self.nodesLayer1,
                                     activation = tf.nn.tanh,)
            dense2 = tf.layers.dense(dense1, units = self.nodesLayer2,
                                     activation = tf.nn.tanh,)
            self.QValues = tf.layers.dense(dense2, units = self.numberActions,)

            self.loss = tf.reduce_mean(tf.square(self.QValues - self.qTarget))
            self.trainOptimizer = tf.train.AdamOptimizer(self.learningRate).minimize(self.loss)
```

DeepQNetwork_TanhSig.py

```
from lib.agents.nn.IDeepQNetwork import IDeepQNetwork
import tensorflow as tf
import numpy as np

class DeepQNetwork_TanhSig(IDeepQNetwork):
    def __init__(self, learningRate, numberActions, inputDimensions, \
                 agentType, neuralNetworkName):
        print("__init__")

        self.nodesLayer1 = 256
        self.nodesLayer2 = 256

        super(DeepQNetwork_TanhSig, self).__init__(learningRate, \
                    numberActions, inputDimensions, \
                    agentType, neuralNetworkName)


    def buildNetwork(self):
        print("...build network...")
        with tf.variable_scope(self.neuralNetworkName):
            print("...building placeholders...")
            self.input = tf.placeholder(tf.float32,
                                        shape = [None, self.inputDimensions],
                                        name = "inputs")
            self.actions = tf.placeholder(tf.float32,
                                          shape = [None, self.numberActions],
                                          name = "actionTaken")
            self.qTarget = tf.placeholder(tf.float32,
                                           shape = [None, self.numberActions],
                                           name = "qValue")

            print("...building layers...")
            flat = tf.layers.flatten(self.input)
            dense1 = tf.layers.dense(flat, units = self.nodesLayer1,
                                     activation = tf.nn.tanh,)
            dense2 = tf.layers.dense(dense1, units = self.nodesLayer2,
                                     activation = tf.nn.sigmoid,)
            self.QValues = tf.layers.dense(dense2, units = self.numberActions,
                                    activation = tf.nn.sigmoid,)

```

```
41
42              self.loss = tf.reduce_mean(tf.square(self.QValues - self.qTarget))
43              self.trainOptimizer = tf.train.AdamOptimizer(self.learningRate).minimize(self.loss)
```

IDeepQNetwork.py

```
1  import os
2  import tensorflow as tf
3  import numpy as np
4
5
6  class IDeepQNetwork(object):
7      basePathCheckpoints = "tmp"
8      def __init__(self, learningRate, numberActions, inputDimensions, agentType, neuralNetworkName)
       :
9
10         self.learningRate = learningRate
11         self.numberActions = numberActions
12         self.neuralNetworkName = neuralNetworkName
13         self.checkpointDirectory = self.__class__.__name__
14         self.agentType = agentType
15         self.inputDimensions = inputDimensions
16         self.buildNetwork()
17         self.session = tf.Session()
18         self.session.run(tf.global_variables_initializer())
19         self.saver = tf.train.Saver()
20         self.checkpointFile = os.path.join(self.basePathCheckpoints, self.agentType, \
21                         self.checkpointDirectory, self.neuralNetworkName, "deepqnet.ckpt")
22
23         self.params = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=self.
       neuralNetworkName)
24
25
26         if not os.path.isdir(self.basePathCheckpoints):
27             os.makedirs(self.basePathCheckpoints)
28
29         self.basePathCheckpointsAgentPath = self.basePathCheckpoints + "/" + self.agentType
30
31         if not os.path.isdir(self.basePathCheckpointsAgentPath):
32             os.makedirs(self.basePathCheckpointsAgentPath)
33
34         self.checkpointsExperimentPath = self.basePathCheckpointsAgentPath + "/" + self.
       checkpointDirectory
35
36         if not os.path.isdir(self.checkpointsExperimentPath):
37             os.makedirs(self.checkpointsExperimentPath)
38
39
40     def buildNetwork(self):
41         raise NotImplementedError("Must be implemented")
42
43     # Has to objective update trainable parameters in the current dqn taking source from dqnSource
```

```
44      def copyNetworkProperties(self, dqnSource):
45          print("...cloning network parameters...")
46          dqnSourceParams = [t for t in tf.trainable_variables() if t.name.startswith(dqnSource.
        neuralNetworkName)]
47          dqnSourceParams = sorted(dqnSourceParams, key=lambda v: v.name)
48          selfParams = [t for t in tf.trainable_variables() if t.name.startswith(self.
        neuralNetworkName)]
49          selfParams = sorted(selfParams, key=lambda v: v.name)
50
51          # Print trainable tf parameters of the two networks
52          # print dqnSourceParams
53          # print selfParams
54
55          update_ops = []
56          for paramSource, paramTarget in zip(dqnSourceParams, selfParams):
57              # Print individual parameters
58              # print paramSource.eval(session = dqnSource.session)
59              # print paramTarget.eval(session = self.session)
60              # ot accomplish the asignation must be used the source session to get their values
61              op = paramTarget.assign(paramSource.eval(session = dqnSource.session))
62              update_ops.append(op)
63
64          self.session.run(update_ops)
65
66      # load a network previously saved
67      def loadCheckpoint(self):
68          print("...Loading checkpoint...")
69          self.saver.restore(self.session, self.checkpointFile)
70
71      # save the network to a file
72      def saveCheckpoint(self):
73          print("...Saving checkpoint...")
74          self.saver.save(self.session, self.checkpointFile)
```

A.2.5. lib/envs

ActionSpace.py

```
1   class ActionSpace(object):
2
3       def RemoveDuplicatedActions(self, duplicate):
4           finalList = []
5           for num in duplicate:
6               if num not in finalList:
7                   finalList.append(num)
8           return finalList
9
10      def __init__(self, actions):
11          actions = self.RemoveDuplicatedActions(actions)
12          self.actions = actions
13          self.n = len(actions)
```

Environment2D.py

```
import numpy as np
import math
import uuid
import copy
import multiprocessing as mp

from lib.envs.IEnvironment import IEnvironment
from lib.envs.ActionSpace import ActionSpace
from lib.envs.Point import Point
import lib.utils.utilities as utilities

# determine if a given point located at (x, y) is contained in a polygon(poly)
def pointIncluded(x, y, poly):
    n = len(poly)
    inside = False

    p1x,p1y = poly[0]
    for i in range(n):
        p2x, p2y = poly[(i + 1) % n]
        if y > min(p1y, p2y):
            if y <= max(p1y, p2y):
                if x <= max(p1x, p2x):
                    if p1y != p2y:
                        xints = (y-p1y)*(p2x-p1x)/(p2y-p1y)+p1x
                    if p1x == p2x or x <= xints:
                        inside = not inside
        p1x, p1y = p2x, p2y
    return inside

# determine if a given point located at (x, y) is outside of the environment boundaries
def pointOutsideEnvironment(x, y, boundaries):
    outside = False

    if boundaries[0] > x\
    or boundaries[1] < x\
    or boundaries[2] > y\
    or boundaries[3] < y:
        outside = True

    return outside

# determine the distance from the actual location to goal
def getDistanceToGoal(p1, p2, goal):
    distance = None

    # determinate zones
    # trajectory
    dy = p2.y - p1.y
    dx = p2.x - p1.x
    if dx == 0:
        ymax = max(p1.y, p2.y)
        ymin = min(p1.y, p2.y)
```

```
53          if ymax >= goal.y and goal.y >= ymin:
54              distance = abs(p2.x - goal.x)
55      elif dy == 0:
56          xmax = max(p1.x, p2.x)
57          xmin = min(p1.x, p2.x)
58          if xmax >= goal.x and goal.x >= xmin:
59              distance = abs(p2.y - goal.y)
60      elif dx != 0:
61
62          m = dy/dx
63          # perpendicular line
64          ml = -1/m
65
66          #line equation y = mx + b
67          # p1
68          b1 = p1.y - ml * p1.x
69          # p2
70          b2 = p2.y - ml * p2.x
71          # goal
72          bg = goal.y - ml * goal.x
73
74          bmax = max(b1, b2)
75          bmin = min(b1, b2)
76          if bmax >= bg and bg >= bmin:
77              distance = getDistanceTrajectoryToGoal(p1, p2, goal)
78
79      if distance == None:
80          dy = goal.y - p2.y
81          dx = goal.x - p2.x
82          distance = math.sqrt(math.pow(dx, 2) + math.pow(dy, 2))
83
84      utilities.printMessage("closest  distance: %s"%(distance), 0)
85      return distance
86
87  # determine the minimum distance from two locations to goal
88  def getDistanceTrajectoryToGoal(p1, p2, goal):
89      dy = goal.y - p1.y
90      dx = goal.x - p1.x
91      dP1G = math.sqrt(math.pow(dx, 2)  + math.pow(dy, 2))
92
93      beta = math.atan2(dy, dx)
94
95      dy = p2.y - p1.y
96      dx = p2.x - p1.x
97      alfa = math.atan2(dy, dx)
98
99      gamma = beta - alfa
100
101     distance = dP1G * math.sin(gamma)
102
103     return abs(distance)
104
105 # calculate the reward using the position of the vehicle, environment boundaries and zones using
        parallel processing
106 def getRewardVehicleByZonesParallel(queue, x, y, boundaries, zones, terminalReward):
107     queue.put(getRewardVehicleByZones(x, y, boundaries, zones, terminalReward))
108
```

```
109 # calculate the reward using the position of the vehicle, environment boundaries and zones
110 def getRewardVehicleByZones(x, y, boundaries, zones, terminalReward):
111     reward = 0
112     done = False
113 
114     if pointOutsideEnvironment(x, y, boundaries) == True:
115         reward = -terminalReward
116         done = True
117         return reward, done # Terminal state found
118 
119     for zoneGroup in zones:
120         zoneType = zoneGroup[0]
121         zonePoligons = zoneGroup[1]
122         for zone in zonePoligons:
123             if reward != -terminalReward:
124                 if pointIncluded(x, y, zone) == True:
125                     if zoneType == "d":
126                         reward = -terminalReward
127                         done = True
128                         return reward, done # Terminal state found
129                     elif zoneType == "w":
130                         reward = -3
131     return reward, done
132 
133 # calculate the reward using the position of the test point, environment boundaries and zones
      using parallel processing
134 def getRewardTestPointByZonesParallel(queue, uuid, x, y, range, boundaries, zones):
135     queue.put(getRewardTestPointByZones(uuid, x, y, range, boundaries, zones))
136 
137 # calculate the reward using the position of the test point, environment boundaries and zones
138 def getRewardTestPointByZones(uuid, x, y, range, boundaries, zones):
139     reward = 0
140     atenuator = 0.10
141 
142     if pointOutsideEnvironment(x, y, boundaries) == True:
143         rewardLargeRange = -5 * atenuator
144         rewardMiddleRange = -7 * atenuator
145         rewardShortRange = -9 * atenuator
146         if(range =="large"):
147             reward = rewardLargeRange
148         elif(range == "middle"):
149             reward = rewardMiddleRange
150         elif (range == "short"):
151             reward = rewardShortRange
152 
153     for zoneGroup in zones:
154         zoneType = zoneGroup[0]
155         zonePoligons = zoneGroup[1]
156         for zone in zonePoligons:
157             if pointIncluded(x, y, zone) == True:
158                 if zoneType == "d":
159                     rewardLargeRange = -5 * atenuator
160                     rewardMiddleRange = -7 * atenuator
161                     rewardShortRange = -9 * atenuator
162                 elif zoneType == "w":
163                     rewardLargeRange = -1 * atenuator
164                     rewardMiddleRange = -3 * atenuator
```

```
165                     rewardShortRange = -5 * atenuator
166
167                 if(range =="large"):
168                     if reward > rewardLargeRange:
169                         reward = rewardLargeRange
170                 elif(range == "middle"):
171                     if reward > rewardMiddleRange:
172                         reward = rewardMiddleRange
173                 elif (range == "short"):
174                     if reward > rewardShortRange:
175                         reward = rewardShortRange
176
177     return uuid, reward
178
179
180 # this class is used to manage the environment 2D
181 class Environment2D(IEnvironment):
182    # The environment is "static" or "dynamic"
183     environmentMode = "dynamic"
184     #just works by environmentMode = "static"
185     scenario = 0
186     initialized = False
187
188     # The position of the vehicle
189     startPosition = None
190     position = None
191
192     # The target point
193     goal = None
194
195     terminalReward = 10000.0
196
197     # Contains the trajectory
198     trajectory = []
199
200     #contains an array with the zones of death
201     warning = []
202
203     #contains an array with the zones of warning
204     death = []
205
206     # Contains an array with the information about test points
207     sensors = []
208
209     observation = None
210     distance = None
211     testPointsLocation = []
212
213     cpuCount = 1
214
215     # initialize the environment
216     def __init__(self, xmin=0, xmax=10, ymin=0, ymax=10, stepLength=0.1, \
217             targetTolerance = .01, environmentMode = "dynamic", scenario = 0):
218         super(Environment2D, self).__init__()
219         utilities.printMessage("__init__")
220         # defines environment mode
221         self.environmentMode = environmentMode
```

```
222         self.scenario = scenario
223         # define size of the world
224         self.xmin = xmin
225         self.ymin = ymin
226         self.xmax = xmax
227         self.ymax = ymax
228 
229         self.boundaries = [xmin, xmax, ymin, ymax]
230 
231         # define vehicle length step
232         self.stepLength = stepLength
233 
234         # Set the tolerance to reach the goal
235         self.targetTolerance = targetTolerance
236 
237         # define action space rigth to left
238         yaw=[-15, -10, -5, -3, -2, -1, -0.5, -0.2, -0.1, 0, 0.1, 0.2, 0.5, 1, 2, 3, 5, 10, 15]
239 
240         # convert to rads
241         yaw = [x * math.pi/180 for x in yaw]
242 
243         actions=[]
244         for da in yaw:
245             actions.append(da)
246 
247         self.actionSpace = ActionSpace(actions)
248 
249         # initializing testPoints
250         self.testPointsInitialize()
251 
252         # reseting the environment
253         self.reset()
254 
255         self.cpuCount = mp.cpu_count()
256 
257     # Initialize test points
258     def testPointsInitialize(self):
259         utilities.printMessage("...test points initializing...")
260         self.sensors = []
261 
262         # LONG RANGE
263         self.sensors.append(self.testPointCreate(10, 0))
264         # long right
265         self.sensors.append(self.testPointCreate(10, -5))
266         # long left
267         self.sensors.append(self.testPointCreate(10, 5))
268 
269         # MIDDLE RANGE
270         # middle right close
271         self.sensors.append(self.testPointCreate(5, -15))
272         # middle left close
273         self.sensors.append(self.testPointCreate(5, 15))
274         # middle right open
275         self.sensors.append(self.testPointCreate(5, -35))
276         # middle left open
277         self.sensors.append(self.testPointCreate(5, 35))
278 
```

```
279         # SHORT RANGE
280         # short right
281         self.sensors.append(self.testPointCreate(2, -55))
282         # short left
283         self.sensors.append(self.testPointCreate(2, 55))
284 
285     # The length property is measured in stepLength
286     def testPointCreate(self, length, angle):
287         utilities.printMessage("...creating test point...")
288 
289         if (length==10):
290             range = "large"
291         elif(length == 5):
292             range = "middle"
293         elif(length == 2):
294             range = "short"
295         else:
296             range = "unknown"
297         sensorReward = 0
298         #print("length: ", length, "range: ", range)
299         testPointIdentifier = str(uuid.uuid4())
300         return [length * self.stepLength, angle * math.pi/180, range, sensorReward,
     testPointIdentifier]
301 
302     # Process a single step and returns information about the next state
303     def step(self, action):
304         utilities.printMessage("step")
305 
306         previousPosition = self.position
307 
308         # The vehicle turns by taking agent action
309         angle = self.actionSpace.actions[action]
310         utilities.printMessage("action: %s"%(action))
311         utilities.printMessage("angle: %s"%(angle))
312         self.position.turn(angle)
313 
314         # The vehicle advance the stepLength
315         self.position.move(self.stepLength)
316 
317         # Add the initial position to the trajectory array
318         self.trajectory.append(self.position)
319 
320         # calculate the reward of this state
321         reward, done = self.calculateReward2(previousPosition, self.position)
322 
323         utilities.printMessage("trajectory length: %s"%(len(self.trajectory)))
324         return (self.getObservationState(), reward, done, None)
325 
326     # Returns the observation of the next state
327     def getObservationState(self):
328         observation = []
329         # calculate a unit vector from actual position to goal
330         positionX = self.position.x
331         positionY = self.position.y
332 
333         targetX = self.goal.x
334         targetY = self.goal.y
```

```
335
336         dx = targetX - positionX
337         dy = targetY - positionY
338
339         h = math.sqrt(pow(dx, 2) + pow(dy, 2))
340
341         self.distance = h
342         utilities.printMessage("straight distance: %s"%(h), 0)
343         # adding a information about the direction to the goal
344         observation.append(dx/h)
345         observation.append(dy/h)
346
347         # adding the orientation of the vehicle
348         observation.append(self.position.getyaw())
349
350         # adding rewards of test points
351         for sensor in self.sensors:
352             sensorReward = sensor[3]
353             observation.append(sensorReward)
354
355         self.observation = observation
356         return observation
357
358     # prints status
359     def printTerminalStatus(self, done, reward):
360         utilities.printMessage("done: %s"%(done), 2)
361         utilities.printMessage("reward: %s"%(reward), 2)
362         if done == True:
363             utilities.printMessage("################################", 0)
364             if reward > 8000:
365                 utilities.printMessage("######### GOAL REACHED #########", 0)
366             elif reward < 8000:
367                 utilities.printMessage("#########   CRASHED    #########", 0)
368             else:
369                 utilities.printMessage("#########    STUCK     #########", 0)
370             utilities.printMessage("################################", 0)
371
372
373
374     # Calculate the reward of the actual position[obsolete]
375     def calculateReward(self, previousPosition, actualPosition):
376         utilities.printMessage("...calculating reward...")
377         reward = 0
378         done = False
379
380         utilities.printMessage("...goal reached?...", 2)
381         distance = getDistanceToGoal(previousPosition, actualPosition, self.goal)
382         if abs(distance) <= abs(self.targetTolerance):
383             reward = self.terminalReward
384             done = True
385
386         #if the state is out of the boundaries
387         if self.xmin > actualPosition.x or self.xmax < actualPosition.x or self.ymin >
     actualPosition.y or self.ymax < actualPosition.y:
388             reward = -self.terminalReward
389             done = True
390
```

```
391         utilities.printMessage("...zones...")
392         #if the state is inside of a death zone
393         if reward == 0:
394             for d in self.death:
395                 if pointIncluded(actualPosition.x, actualPosition.y, d):
396                     reward = -self.terminalReward
397                     done = True
398 
399         #if the state is inside of a warning zone
400         if reward == 0:
401             for w in self.warning:
402                 if pointIncluded(actualPosition.x, actualPosition.y, w):
403                     reward = -3
404 
405         #for each step increases the negative reward
406         if done == False:
407             reward += -0.01
408 
409         utilities.printMessage("vehicle reward: %d"%(reward), 8)
410         utilities.printMessage("...sensors zones...", 2)
411         if(reward != -self.terminalReward and reward != self.terminalReward):
412             z = []
413             z.append(["d", self.death])
414             z.append(["w", self.warning])
415             atenuator = 10
416             self.testPointsLocation = []
417             for sn in self.sensors:
418                 sensorReward = 0
419                 rewardLargeRange = -5.0 / atenuator
420                 rewardMiddleRange = -7.0 / atenuator
421                 rewardShortRange = -9.0 / atenuator
422                 snx, sny = self.getPointTest(actualPosition, sn)
423                 range = sn[2]
424 
425                 self.testPointsLocation.append([snx, sny, sn[2]])
426 
427                 if self.xmin > snx or self.xmax < snx or self.ymin > sny or self.ymax < sny:
428                     if(range =="large"):
429                         sensorReward = rewardLargeRange
430                     elif(range == "middle"):
431                         sensorReward = rewardMiddleRange
432                     elif (range == "short"):
433                         sensorReward = rewardShortRange
434 
435                 for zoneGroup in z:
436 
437                     zoneType = zoneGroup[0]
438                     if(zoneType == "d"):
439                         rewardLargeRange = -5.0 / atenuator
440                         rewardMiddleRange = -7.0 / atenuator
441                         rewardShortRange = -9.0 / atenuator
442                     elif (zoneType == "w"):
443                         rewardLargeRange = -1.0 / atenuator
444                         rewardMiddleRange = -3.0 / atenuator
445                         rewardShortRange = -5.0 / atenuator
446                     zones = zoneGroup[1]
447                     for d in zones:
```

```
448                     if pointIncluded(snx, sny, d):
449                         if(range =="large"):
450                             if sensorReward > rewardLargeRange:
451                                 sensorReward = rewardLargeRange
452                         elif(range == "middle"):
453                             if sensorReward > rewardMiddleRange:
454                                 sensorReward = rewardMiddleRange
455                         elif (range == "short"):
456                             if sensorReward > rewardShortRange:
457                                 sensorReward = rewardShortRange
458             reward = reward + sensorReward
459             sn[3] = sensorReward
460 
461     self.printTerminalStatus(done, reward)
462     utilities.printMessage("sn: %s"%(self.sensors), 2)
463 
464     return (reward, done)
465 
466 # Calculate the reward of the actual position improved over the first version
467 def calculateReward2(self, previousPosition, actualPosition):
468     utilities.printMessage("...calculating reward 2...")
469     reward = 0
470     done = False
471 
472     utilities.printMessage("...goal reached 2?...", 2)
473     distance = getDistanceToGoal(previousPosition, actualPosition, self.goal)
474     if abs(distance) <= abs(self.targetTolerance):
475         reward = self.terminalReward
476         done = True
477 
478     environmentZones = []
479     environmentZones.append(["d", self.death])
480     environmentZones.append(["w", self.warning])
481 
482     if done == False:
483         reward, done = getRewardVehicleByZones(actualPosition.x, actualPosition.y, self.
    boundaries, environmentZones, self.terminalReward)
484 
485     #for each step increases the negative reward
486     if done == False:
487         reward += -0.01
488 
489     utilities.printMessage("vehicle reward2: %d"%(reward), 8)
490     utilities.printMessage("...sensors zones2...")
491     if(done == False):
492         self.testPointsLocation = []
493         for sn in self.sensors:
494             snx, sny = self.getPointTest(actualPosition, sn)
495             range = sn[2]
496             self.testPointsLocation.append([snx, sny, sn[2]])
497 
498             uuid, sensorReward = getRewardTestPointByZones(sn[4], snx, sny, range, self.
    boundaries, environmentZones)
499 
500             reward = reward + sensorReward
501             sn[3] = sensorReward
502 
```

```
503         self.printTerminalStatus(done, reward)
504         utilities.printMessage("sn: %s"%(self.sensors))
505         return (reward, done)
506 
507     # Calculate the reward of the actual position can be used to parallel processing
508     def calculateReward3(self, previousPosition, actualPosition):
509         utilities.printMessage("...calculating reward 3...")
510         reward = 0
511         done = False
512 
513         utilities.printMessage("...goal reached 3?...", 8)
514         distance = getDistanceToGoal(previousPosition, actualPosition, self.goal)
515         if abs(distance) <= abs(self.targetTolerance):
516             reward = self.terminalReward
517             done = True
518 
519         environmentZones = []
520         environmentZones.append(["d", self.death])
521         environmentZones.append(["w", self.warning])
522 
523         if done == False:
524             reward, done = getRewardVehicleByZones(actualPosition.x, actualPosition.y, self.
        boundaries, environmentZones, self.terminalReward)
525 
526         #for each step increases the negative reward
527         if done == False:
528             reward = -0.01
529 
530         utilities.printMessage("vehicle reward2: %d"%(reward))
531         utilities.printMessage("...sensors zones3...")
532         if(done == False):
533             processes = []
534             queue = mp.Queue()
535             self.testPointsLocation = []
536             for sn in self.sensors:
537                 snx, sny = self.getPointTest(actualPosition, sn)
538                 range = sn[2]
539                 self.testPointsLocation.append([snx, sny, sn[2]])
540 
541                 process = mp.Process(target = getRewardTestPointByZonesParallel, args = (queue, sn
        [4], snx, sny, range, self.boundaries, environmentZones, ))
542                 processes.append(process)
543 
544             [p.start() for p in processes]
545             [p.join() for p in processes]
546 
547             while queue.empty() is False:
548                 result = queue.get()
549                 sensorIdentifier = result[0]
550                 sensorReward = result[1]
551                 #find sensor and assign reward
552                 testPoint = filter(lambda x: x[4] == sensorIdentifier, self.sensors)
553                 if testPoint != []:
554                     testPoint[0][3] = result[1]
555                     reward = reward + sensorReward
556 
557         self.printTerminalStatus(done, reward)
```

```
558         return (reward, done)
559 
560     # get location of the test point
561     def getPointTest(self, position, sensor):
562         sx=position.x
563         sy=position.y
564         syaw=position.yaw
565 
566         sensorlength = sensor[0]
567         sensorAngle = sensor[1]
568 
569         ps = Point(sx, sy, syaw)
570         ps.turn(sensorAngle)
571         ps.move(sensorlength)
572 
573         return ps.x, ps.y
574 
575     # performs a reset of the environment
576     def reset(self):
577         utilities.printMessage("reset environment", 1)
578         if self.initialized == None:
579             self.initialized = False
580         if self.environmentMode == "dynamic" or (self.environmentMode == "static" and self.
    initialized == False):
581             self.initializeZones()
582             self.initializeStartPoint()
583             self.initializeGoalPoint()
584             self.initialized = True
585 
586         if self.environmentMode == "static":
587             if self.scenario == None:
588                 self.scenario = 1
589             self.loadScenario(self.scenario)
590 
591         self.position = None
592 
593         self.position = copy.deepcopy(self.startPosition)
594 
595         # Initialize the trajectory array whit the initial position
596         self.trajectory = []
597         self.trajectory.append(self.position)
598 
599         utilities.printMessage("position: x: %s, y: %s"%(self.position.x, self.position.y))
600         utilities.printMessage("goal: x: %s, y: %s"%(self.goal.x, self.goal.y))
601         return self.getObservationState()
602 
603     # load a given scenario
604     def loadScenario(self, scenario):
605         utilities.printMessage("loaging scenario", 2)
606         xmin = self.xmin
607         ymin = self.ymin
608         xmax = self.xmax
609         ymax = self.ymax
610 
611         dx = xmax - xmin
612         dy = ymax - ymin
613 
```

```
614         self.startPosition = Point(xmin + (0.1 * dx), ymin + (0.1 * dy), 0)
615         self.goal = Point(xmin + (0.9 * dx), ymin + (0.9 * dy), 0)
616 
617         self.death = []
618         self.warning = []
619 
620         if scenario == 2:
621             utilities.printMessage("scenario 2")
622             if(dy < dx):
623                 lz = dy / 3
624             else:
625                 lz = dx / 3
626             zone = self.assignZonePoints(xmin + (0.5 * dx), ymin + (0.5 * dy), lz, 9)
627             self.death.append(zone)
628 
629         elif scenario == 3:
630             utilities.printMessage("scenario 3")
631             if(dy < dx):
632                 lz = dy / 5
633             else:
634                 lz = dx / 5
635             zone = self.assignZonePoints(xmin + (0.3 * dx), ymin + (0.7 * dy), lz, 9)
636             self.death.append(zone)
637             zone = self.assignZonePoints(xmin + (0.8 * dx), ymin + (0.8 * dy), 0.1 * lz, 9)
638             self.death.append(zone)
639             zone = self.assignZonePoints(xmin + (0.7 * dx), ymin + (0.3 * dy), lz, 9)
640             self.warning.append(zone)
641 
642     # initialize the start point
643     def initializeStartPoint(self):
644         utilities.printMessage("...initializing start point...")
645         createStarPoint = False
646         while not createStarPoint:
647             x0 = np.random.uniform(self.xmin, self.xmax)
648             y0 = np.random.uniform(self.ymin, self.ymax)
649 
650             included = False
651 
652             for zoneDeath in self.death:
653                 included = included | pointIncluded(x0, y0, zoneDeath)
654 
655             for zoneWarning in self.warning:
656                 included = included | pointIncluded(x0, y0, zoneWarning)
657 
658             if(included == True):
659                 createStarPoint = False
660             else:
661                 createStarPoint = True
662 
663         yaw0 = np.random.uniform(0, 2 * math.pi)
664         self.startPosition = Point(x0, y0, yaw0)
665 
666     # intialize the goal point
667     def initializeGoalPoint(self):
668         utilities.printMessage("...initializing goal point...")
669         createGoalPoint = False
670         while not createGoalPoint:
```

```
671             xG = np.random.uniform(self.xmin, self.xmax)
672             yG = np.random.uniform(self.ymin, self.ymax)
673 
674             included = False
675 
676             for zoneDeath in self.death:
677                 included = included | pointIncluded(xG, yG, zoneDeath)
678 
679             for zoneWarning in self.warning:
680                 included = included | pointIncluded(xG, yG, zoneWarning)
681 
682             if(included == True):
683                 createGoalPoint = False
684             else:
685                 createGoalPoint = True
686 
687         self.goal = Point(xG, yG)
688 
689     # Initialize environment warning and death zones
690     def initializeZones(self):
691         utilities.printMessage("...initializing environment zones...")
692 
693         self.death = []
694         self.warning = []
695         dx = self.xmax - self.xmin
696         dy = self.ymax - self.ymin
697 
698         if(dy < dx):
699             dx = dy / 5
700         else:
701             dx = dx / 5
702 
703         numberDeathZones = 2
704         numberWarningZones = 3
705 
706         for zone in range(numberDeathZones):
707             dx2 = np.random.uniform(0, dx)
708             self.death.append(self.generateZone(dx2))
709 
710         for zone in range(numberWarningZones):
711             dx2 = np.random.uniform(0, dx)
712             self.warning.append(self.generateZone(dx2))
713 
714     # Generate a polygon which cannot be included in another zone
715     def generateZone(self, length):
716         zone = []
717         xZ = 0
718         yZ = 0
719         createZone = False
720         while not createZone:
721             xZ = np.random.uniform(self.xmin, self.xmax)
722             yZ = np.random.uniform(self.ymin, self.ymax)
723 
724             included = False
725 
726             for zoneDeath in self.death:
727                 included = included | pointIncluded(xZ, yZ, zoneDeath)
```

```
728
729             for zoneWarning in self.warning:
730                 included = included | pointIncluded(xZ, yZ, zoneWarning)
731
732             if(included == True):
733                 createZone = False
734             else:
735                 createZone = True
736
737             utilities.printMessage("zones: %s"%(len(self.death) + len(self.warning)))
738             utilities.printMessage("included: %s"%(included))
739             utilities.printMessage("createZone: %s"%(createZone))
740         edges = np.random.randint(3, 10)
741         zone = self.assignZonePoints(xZ, yZ, length, edges)
742
743         return zone
744
745     # Generate polygon points
746     def assignZonePoints(self, xZ, yZ, length, edges):
747         zone = []
748         for edge in range(edges):
749             a = 2 * math.pi / edges
750             xi = xZ + length * math.sin(edge * a)
751             yi = yZ + length * math.cos(edge * a)
752             zone.append([xi, yi])
753
754         return zone
```

IEnvironment.py

```
1  import lib.Settings as settings
2
3  ### Interface
4  class IEnvironment(object):
5
6      def __init__(self):
7          settings.init()
8
9      def reset(self):
10         raise NotImplementedError("Inheriting classes must override reset.")
11
12     def actions(self):
13         raise NotImplementedError("Inheriting classes must override actions.")
14
15     def step(self):
16         raise NotImplementedError("Inheriting classes must override step")
```

Point.py

```
import math

class Point(object):
    def __init__(self, x = 0, y = 0, yaw = 0):
        self.x = x
        self.y = y
        self.yaw = yaw

    # turns the orientation of the point by angle units
    def turn(self, angle):
        self.yaw += angle

        if (self.yaw > 2 * math.pi):
            self.yaw -= 2 * math.pi
        elif (self.yaw < - 2 * math.pi):
            self.yaw += 2 * math.pi

    # moves the point length units in the orientation direction
    def move(self, length):
        self.x += length * math.cos(self.yaw)
        self.y += length * math.sin(self.yaw)

    # prints the point information
    def info(self):
        print("x: %s y: %s yaw: %s"%(self.x, self.y, self.yaw))

    # returns the orientation of the point
    def getyaw(self):
        while (self.yaw < -math.pi) or (self.yaw > math.pi):
            if (self.yaw > math.pi):
                self.yaw -= 2 * math.pi
            elif (self.yaw < -math.pi):
                self.yaw += 2 * math.pi
        return self.yaw
```

A.2.6. lib/utils

---

csv2graph.py

```
import csv
from matplotlib import pyplot as plt
from matplotlib.gridspec import GridSpec
import pandas as pd
import math

class csv2graph(object):
    def __init__(self, filePath):
        print "__Init__"
        self.filePath = filePath
```

```
11
12      def createGraphic(self):
13          print "...loading file content..."
14          episodeNumber = []
15          episodeReward = []
16          episodeLength = []
17          with open(self.filePath) as csvfile:
18              readCSV = csv.reader(csvfile, delimiter=";")
19              header = True
20              for row in readCSV:
21                  if(header == True):
22                      header = False
23                  else:
24                      episodeNumber.append(int(row[0]))
25                      episodeReward.append(float(row[1]))
26                      episodeLength.append(int(row[2]))
27
28          print "...initializing graph..."
29
30          maxNumberEpisodes = max(episodeNumber)
31          maxNumberSteps = max(episodeLength)
32
33          self.fig = plt.figure(figsize=(10, 5))
34          self.gs = GridSpec(2, 1)
35
36          self.axEpisodeReward = plt.subplot(self.gs[0, 0])
37
38          self.axEpisodeLength = plt.subplot(self.gs[1, 0])
39
40          self.axEpisodeReward.set_xlim(1, maxNumberEpisodes)
41          self.axEpisodeReward.set_ylim(-12000, 12000)
42          self.axEpisodeReward.yaxis.set_label_position("right")
43          self.axEpisodeReward.set_ylabel("Reward")
44
45          self.axEpisodeLength.set_xlim(1, maxNumberEpisodes)
46          self.axEpisodeLength.set_ylim(0, maxNumberSteps * 1.1)
47          self.axEpisodeLength.yaxis.set_label_position("right")
48          self.axEpisodeLength.set_ylabel("Length")
49          self.axEpisodeLength.set_xlabel("Episode")
50
51          print "...processing graphic..."
52          size = len(episodeNumber)
53
54          self.axEpisodeReward.plot(episodeNumber, episodeReward, marker="", color="gray")
55          self.axEpisodeLength.plot(episodeNumber, episodeLength, marker="", color="gray")
56          if(size > 100):
57              sizeGroup = size / 100
58              sizeGroup = math.floor(sizeGroup / 5.0)
59              if(sizeGroup == 0):
60                  sizeGroup = 1
61              size = int(sizeGroup) * 5
62
63              episodeReward_smoothed = pd.Series(episodeReward).rolling(size, min_periods=size).mean
    ()
64              episodeLength_smoothed = pd.Series(episodeLength).rolling(size, min_periods=size).mean
    ()
65              self.axEpisodeReward.plot(episodeNumber, episodeReward_smoothed, marker="", color="b")
```

```
66              self.axEpisodeLength.plot(episodeNumber, episodeLength_smoothed, marker="", color="b")
67
68              # self.axEpisodeReward.set_title("Episode information (red line smoothed over window %
    d)"% size)
69              # self.axEpisodeReward.set_ylabel("Reward (Smoothed)")
70              # self.axEpisodeLength.set_ylabel("Length (Smoothed)")
71              self.axEpisodeLength.set_xlabel("Episode(blue line smoothed over window %d)"% size)
72          else:
73              self.axEpisodeReward.plot(episodeNumber, episodeReward, marker="", color="blue")
74              self.axEpisodeLength.plot(episodeNumber, episodeLength, marker="", color="blue")
75
76          print "...saving image..."
77          self.fig.savefig(self.filePath + ".png", bbox_inches="tight")
78          print "...finish..."
```

---

utilities.py

```
1  import lib.Settings as settings
2
3  # this function will print a debug message if this behavior is enabled
4  def printMessage(message, level = 9):
5      # if the value is not defined then False is assigned
6      if settings.printDebugMessage == None:
7          settings.printDebugMessage = False
8
9      # if the value is not defined then 0 is assigned
10     if settings.printDebugLevel == None:
11         settings.printDebugLevel = 0
12
13     # print the message if it's enabled and fulfill the level required
14     if settings.printDebugMessage == True and settings.printDebugLevel >= level:
15         print(message)
```

A.2.7. test

En esta carpeta se almacenan las pruebas unitarias y/o de integración elaboradas, sin embargo, no se incluyen ya que no se consideran relevantes.

A.2.8. tmp

En la carpeta "double" se almacenan los datos resultantes del entrenamiento de los agentes Doble DQN, indicando la topología de red neuronal como por ejemplo "DeepQNetwork_SigTanh", y en ella se guarda la configuración de cada una de las redes neuronales empleadas "qEval" y "qTarget".

En la carpeta “rules” se almacená el archivo xml que tendrá la configuración del agente experto, en la sección B.2 se muestra un archivo limitado a modo de ejemplo.

En la carpeta “single” se guardarán los datos resultantes del entrenamiento de la red neuronal indicando la topología empleada, en la carpeta “qEval” se utiliza para el agente DQN y la carpeta “neuralNetwork” se utiliza para la red neuronal usada en el agente experto.

Apéndice B

Archivos resultantes

B.1. Archivo de definición

El siguiente archivo muestra la definición del xml que se obtendrá al final del entrenamiento del sistema experto.

---

xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="ConfigAgent">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="info"/>
        <xs:element ref="Rules"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="info">
    <xs:complexType>
      <xs:attribute name="groups" use="required" type="xs:integer"/>
      <xs:attribute name="partitions" use="required" type="xs:integer"/>
      <xs:attribute name="rules" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Rules">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="Rule"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Rule">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="Condition"/>
```

```
      </xs:sequence>
      <xs:attribute name="action" use="required" type="xs:integer"/>
      <xs:attribute name="id" use="required"/>
      <xs:attribute name="name" use="required"/>
      <xs:attribute name="priority" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Condition">
    <xs:complexType>
      <xs:attribute name="fulfilCondition" use="required" type="xs:NCName"/>
      <xs:attribute name="operation" use="required" type="xs:NCName"/>
      <xs:attribute name="value" use="required" type="xs:integer"/>
      <xs:attribute name="variable" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

B.2. Archivo XML

El siguiente archivo muestra secciones del archivo que se generará a partir del entrenamiento, dicho archivo contiene las reglas que serán usadas por el agente experto.

---

xml

```
<ConfigAgent>
    <info groups="3600" partitions="9" rules="540"></info>
    <Rules>
        <Rule id="ca746....50df133" name="ca746....50df133" action="18" priority="-1000">
            <Condition variable ="deltayaw" operation="le" value="400" fulfilCondition="True"/>
            <Condition variable ="deltayaw" operation="ge" value="1" fulfilCondition="True"/>
        </Rule>
        <Rule id="66cad....185a4f5" name="66cad....185a4f5" action="1" priority="0">
            <Condition variable ="deltayaw" operation="le" value="399" fulfilCondition="True"/>
            <Condition variable ="deltayaw" operation="ge" value="2" fulfilCondition="True"/>
            ...
            <Condition variable ="middle35L" operation="le" value="0" fulfilCondition="True"/>
            <Condition variable ="middle35L" operation="ge" value="0" fulfilCondition="True"/>
            ...
            <Condition variable ="short55R" operation="le" value="0" fulfilCondition="True"/>
            <Condition variable ="short55R" operation="ge" value="0" fulfilCondition="True"/>
        </Rule>
        ......
        <Rule id="cdd9f....9b45ba1" name="cdd9f....9b45ba1" action="17" priority="0">
            <Condition variable ="deltayaw" operation="le" value="449" fulfilCondition="True"/>
            <Condition variable ="deltayaw" operation="ge" value="415" fulfilCondition="True"/>
            <Condition variable ="long0" operation="le" value="0" fulfilCondition="True"/>
            <Condition variable ="long0" operation="ge" value="0" fulfilCondition="True"/>
            <Condition variable ="long05L" operation="le" value="0" fulfilCondition="True"/>
            ....
            <Condition variable ="short55L" operation="le" value="0" fulfilCondition="True"/>
```

```
            <Condition variable ="short55L" operation="ge" value="0" fulfilCondition="True"/>
            <Condition variable ="short55R" operation="le" value="0" fulfilCondition="True"/>
            <Condition variable ="short55R" operation="ge" value="0" fulfilCondition="True"/>
        </Rule>
        ......
        <Rule id="94bd0....243f4fa" name="94bd0....243f4fa" action="15" priority="0">
            <Condition variable ="deltayaw" operation="le" value="2359" fulfilCondition="True"/>
            <Condition variable ="deltayaw" operation="ge" value="2110" fulfilCondition="True"/>
            <Condition variable ="long0" operation="le" value="0" fulfilCondition="True"/>
            <Condition variable ="long0" operation="ge" value="0" fulfilCondition="True"/>
            ....
            <Condition variable ="short55L" operation="le" value="0" fulfilCondition="True"/>
            <Condition variable ="short55L" operation="ge" value="0" fulfilCondition="True"/>
            <Condition variable ="short55R" operation="le" value="0" fulfilCondition="True"/>
            <Condition variable ="short55R" operation="ge" value="0" fulfilCondition="True"/>
        </Rule>
    </Rules>
</ConfigAgent>
```