ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS

INNOVACIÓN PARA LA EXCELENCIA

DEPARTAMENTO DE ELÉCTRICA Y ELECTRÓNICA

CARRERA DE INGENIERÍA EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

PROYECTO DE INVESTIGACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO EN INGENIERÍA

TEMA: CONTROL NEURONAL DEL ROBOT MÓVIL PIONEER P3-DX MEDIANTE UN PERCEPTRÓN MULTICAPA IMPLEMENTADO EN MATLAB

AUTOR: SANTILLÁN ROBALINO, PAÚL DANILO

DIRECTOR: ING. PROAÑO ROSERO, VÍCTOR

SANGOLQUÍ

2015

Declaración de responsabilidad

UNIVERSIDAD DE LAS FUERZAS ARMADAS - ESPE

INGENIERÍA EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

DECLARACIÓN DE RESPONSABILIDAD

PAÚL DANILO SANTILLÁN ROBALINO

DECLARO QUE:

El proyecto de investigación, denominado "Control Neuronal del Robot Móvil Pioneer P3–DX mediante un perceptrón multicapa implementado en Matlab", es el resultado de una profunda investigación científica, en la que se ha respetado el derecho intelectual de terceros, conforme a las citas introducidas y las referencias bibliográficas.

Consecuentemente declaro que este proyecto es de mi autoría.

En virtud de esta declaración, me responsabilizo del contenido integro, y de la veracidad y alcance científico del proyecto de investigación en mención.

Sangolquí, 6 de julio del 2015

Paúl Danilo Santillán Robalino

Autorización de publicación

UNIVERSIDAD DE LAS FUERZAS ARMADAS - ESPE

INGENIERÍA EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

AUTORIZACIÓN DE PUBLICACIÓN

Yo, Paúl Danilo Santillán Robalino,

Autorizo a la Universidad de las Fuerzas Armadas - ESPE la publicación, en la biblioteca virtual de la Institución, del proyecto "Control Neuronal del Robot Móvil Pioneer P3–DX mediante un perceptrón multicapa implementado en Matlab", cuyo contenido, ideas y criterios son de mi exclusiva responsabilidad y autoría.

Sangolquí, 6 de julio del 2015

Paúl Danilo Santillán Robalino

Certificado de tutoría

UNIVERSIDAD DE LAS FUERZAS ARMADAS - ESPE

INGENIERÍA EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

CERTIFICADO DE TUTORÍA

Ing. Víctor Proaño, MSc.

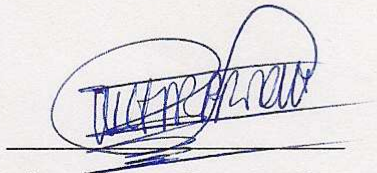
CERTIFICA

Que el proyecto titulado "Control Neuronal del Robot Móvil Pioneer P3–DX mediante un perceptrón multicapa implementado en Matlab" desarrollado en su totalidad por Paúl Danilo Santillán Robalino, ha sido guiado y revisado periódicamente y cumple normas estatutarias establecidas por la universidad en el Reglamento de Estudiantes.

Por tratarse de un trabajo de investigación, recomienda su publicación.

El mencionado trabajo consta de un documento empastado y un disco compacto, el cual contiene los archivos en formato portátil de Acrobat (pdf). Autoriza a Paúl Danilo Santillán Robalino que le entregue al Ingeniero Luis Orozco, en su calidad de Coordinador de la Carrera.

Sangolquí, 6 de julio del 2015



Ing. Víctor Proaño R.
DIRECTOR

DEDICATORIA

Este proyecto está dedicado a mi madre Judith…
alimento de mi alma y
luz de mi camino.

Además está dedicado a aquel estudiante
resuelto a continuar mi investigación.

AGRADECIMIENTOS

En primer lugar, y aunque mis palabras se quedan cortas, a mis padres Judith y Luis por su incondicional amparo espiritual y material. A mi madre por sus interminables muestras de amor, y a mi padre por ser ejemplo de honestidad y nobleza. Además a mi hermana Diana por su tolerancia y paciencia.

Mi más sincero agradecimiento al Ingeniero Víctor Proaño por permitirme trabajar y aprender a su lado, por sus valiosas contribuciones en el desarrollo, por todo su apoyo científico – técnico y también por su apoyo moral.

Además un especial agradecimiento al Ingeniero Paúl Ayala por su interés en mi investigación, por compartirme sus conocimientos, y por sus correcciones y críticas.

INDICE GENERAL

INDICE DE FIGURAS

INDICE DE TABLAS

INDICE DE ECUACIONES

RESUMEN

En el presente proyecto se desarrolló un Controlador Neuronal sobre la Plataforma Robótica Móvil Pioneer P3-DX, para que ésta tenga comportamiento inteligente y un alto grado de autonomía. El controlador neuronal es capaz de resolver un problema de Control de Trayectoria en una pista, dentro de la que el robot evade las paredes y evita las colisiones, usando su arreglo de sensores ultrasónicos SONAR y sus dos motores DC reversibles. El esquema de control neuronal empleado es de tipo adaptativo directo basado en un modelo de referencia del entorno, simulado en MobileSim. El control de trayectoria se probó usando el simulador. El controlador neuronal es diseñado con un perceptrón multicapa, una red neuronal alimentada hacia adelante, creada, entrenada y simulada con el Neural Network Toolbox de Matlab R2014a. La red fue entrenada con aprendizaje supervisado, mediante tres métodos de entrenamiento de retropropagación por descenso de gradiente, que aproximan el cálculo del gradiente de segundo orden: el método de Levenberg-Marquardt 'trainlm' y los métodos Quasi-Newton 'trainbfg' y 'trainoss'. Estos son comparados para determinar cuál tiene mejor rendimiento. El entrenamiento de la red es considerado una optimización numérica de un sistema no lineal, cuyos patrones de entrenamiento son adquiridos en la pista. Para conseguir la transferencia de datos entre Matlab, lenguaje en el que es programado el neurocontrolador, y C++, lenguaje en el que están precompiladas las librerías ARIA del robot, se implementa una Interfaz Matlab ARIA mediante otra herramienta de Matlab, los archivos MEX.

PALABRAS CLAVES:

CONTROLADOR NEURONAL

CONTROL DE TRAYECTORIA

PLATAFORMA ROBÓTICA MÓVIL PIONEER P3-DX

SISTEMA NO LINEAL

INTERFAZ MATLAB ARIA – ARCHIVOS MEX

ABSTRACT

In this Project, a neuro-controller for mobile robotics platform Pioneer P3-DX is developed so that it has intelligent behavior and a high degree of autonomy. The neuro-controller is able to solve a trajectory control problem on a track, inside which the robot evades the walls and avoids collisions, using his forward-facing ultrasonic sensors SONAR and his two reversible DC motors. The employed neuro-control model is of direct adaptive control type based on an environment reference model, simulated in MobileSim. The trajectory control was tested using the simulator. The neuro-controller is designed with a multilayer perceptron, a feedforwardnet: created, trained and simulated in a Neural Network Toolbox of Matlab R2014a. The neural network is trained in supervised learning by using back-propagation training methods that computing gradient descent and updating an approximate second-order gradient: the Levenberg-Marquardt method ‘trainlm’ and Quasi-Newton methods ‘trainbfg’ and ‘trainoss’. They are evaluated to determine which performs better. The neural network training is formulated as a numeric optimization of a nonlinear system, and training patterns are acquired inside the track. To data transfer between MATLAB, the used language in the neuro-controller programming, and C++, the language that ARIA libraries of robot are precompiled, an MATLAB ARIA interface is implemented using MEX-Files, another MATLAB’s tool.

1. INTRODUCCION

1.1. Antecedente

El presente proyecto es una continuación del trabajo previo realizado en tres Proyectos de Grado para la obtención del Título en Ingeniería, en la Carrera de Ingeniería en Electrónica, Automatización y Control, del Departamento de Eléctrica y Electrónica de la ESPE.

El primer proyecto se titula “Desarrollo de Aplicaciones y Documentación de las Plataformas Robóticas Pioneer P3-DX y Pioneer P3-AT”, fue desarrollado en el año 2010 por Jimena Morales y Daniel Jaramillo. En él se analiza y documenta el funcionamiento y el uso de la Pioneer P3-DX, su estructura física y su interface ARIA (Advanced Robotics Interface for Applications). Por tal motivo es el primer acercamiento que se tiene al robot.

El segundo proyecto “Evolución Artificial y Robótica Autónoma desarrollada en el Robot P3-DX con aproximación basada en comportamientos” fue realizado por Marco Flores y Andrés Proaño en el año 2013. De éste, la demostración de robótica autónoma sobre la plataforma es una guía para el desarrollo del presente proyecto. Además, los ejemplos incluidos de programación ARIA en Visual C++ complementan los conocimientos del lenguaje C++.

Finalmente el tercer proyecto fue presentado en el año 2013 por Diego Guffanti y se llama “Control Remoto por Voz del Robot Pioneer P3-DX”. Analiza la Integración de dos Lenguajes de Programación: C++ y Matlab.

C++ es el lenguaje de pre-compilación de las librerías de ARIA. Matlab es un lenguaje que ofrece alta velocidad en el análisis numérico y un *Toolbox* exclusivo para el diseño de redes neuronales. Características que lo convierten en el software elegido para la elaboración del proyecto actual.

De este modo, se demuestra que los tres proyectos señalados son la base y punto de partida para este proyecto.

1.2. Justificación e Importancia

Este proyecto es un aporte al Departamento de Eléctrica y Electrónica, y a la carrera de Ingeniería en Electrónica, Automatización y Control no solo en el campo de la investigación sino también en el de la docencia.

El departamento tiene como tarea primaria el continuo desarrollo investigativo. Este proyecto encaja en dos de sus líneas de investigación: en la de Automatización y Control, y en la de Robótica, abarcando temas específicos del Control Neuronal y de la Robótica Autónoma.

En el departamento hay proyectos previos que han logrado progresos en el control de una plataforma robótica móvil, como el Control Remoto por Voz para la Pioneer P3-DX, más ninguno ha conseguido un grado de autonomía superior.

El proyecto actual es el primero en conseguirlo mediante el desarrollo de un Controlador Inteligente, puntualmente de un Controlador Neuronal para la mencionada plataforma. Convirtiéndolo dentro del departamento en el proyecto pionero en juntar este par de temas de creciente interés, de permanente estudio, de continuo desarrollo y de intensa aplicación en la ingeniería.

En el ámbito de la docencia, este documento sirve de Guía de Práctica para los estudiantes de la Carrera, especialmente para aquellos que cursan

las asignaturas de Control Inteligente y Robótica. Ellos pueden repetir el proyecto sin complicaciones y en profundo entendimiento, porque este es fundamentado en los conocimientos teóricos impartidos en las asignaturas, y es documentado paso a paso y en detalle.

Los estudiantes pueden visualizar como se pone en práctica los conocimientos recibidos en aplicaciones de gran demanda en la actualidad, como: la exploración, la navegación y la vigilancia robótica; y además pueden encontrar varios temas para futuros trabajos en estas líneas de investigación.

Adicionalmente se debe resaltar el estudio que se realiza de la Interfaz Matlab Aria con Archivos MEX. Este conocimiento es compartido con los estudiantes, para que ellos tengan sustento y se animen a generar sus propias interfaces.

Por todo lo antes dicho, se puede afirmar que el desarrollo del proyecto deja dos grandes beneficiados: el Departamento y los estudiantes, principalmente los pertenecientes a nuestra Carrera.

1.3. Alcance del Proyecto

En el proyecto actual se desarrolla un Controlador Neuronal sobre la Plataforma Robótica Móvil Pioneer P3-DX, para que ésta tenga comportamiento inteligente y un alto grado de autonomía.

Para demostrar lo antes dicho, se hace un Control de Trayectoria de la plataforma, una aplicación tradicional para una Red Neuronal Artificial. También cabe señalar, que el control de trayectoria es base para el desarrollo de otras aplicaciones, como: la exploración, la navegación y la vigilancia robótica.

Para el diseño del Controlador Neuronal se utiliza el Neural Network

Toolbox de Matlab, ya que esta herramienta dispone de los comandos apropiados para la creación, el entrenamiento y la simulación de una red, en este caso un perceptrón multicapa.

Para conseguir la transferencia de datos entre C++ (lenguaje en el que están precompiladas las librerías ARIA del robot) y Matlab (lenguaje propietario del Neural Network Toolbox) se implementa una Interfaz Matlab ARIA mediante otra herramienta muy útil de Matlab, los MEX-Files.

También se hace una evaluación para determinar el algoritmo de entrenamiento rápido, que entrega mejores resultados al entrenar la red. Para esto se genera un esquema, en el que se compara el rendimiento de los siguientes tres métodos: Quasi-Newton *'trainbfg',* Quasi-Newton *'trainoss',* y Levenberg-Marquardt *'trainlm*'

Adicionalmente se mencionan brevemente los trabajos futuros que se podrían realizar en base a este proyecto, para así fortalecer las mencionadas Líneas de Investigación.

1.4. Objetivos

1.4.1. General

Desarrollar un Controlador Neuronal, capaz de ejecutar un Control de Trayectoria sobre el robot móvil Pioneer P3-DX, basado en un perceptrón multicapa entrenado mediante método descenso de gradiente de segundo orden, empleando las herramientas de Matlab: "Neural Network Toolbox" y "MEX Files".

1.4.2. Específicos

- Diseñar un Controlador Neuronal, basado en el esquema de control neuronal indirecto: basado en un modelo de proceso generado con

una red neuronal, utilizando la herramienta "Neural Network Toolbox".

- Utilizar la herramienta de Matlab "MEX Files" para crear una Interfaz de comunicación entre los lenguajes de programación Aria y Matlab, para transferir datos entre los algoritmos de control desarrollados en Matlab y la plataforma robótica móvil Pioneer P3-DX.

1.5. Control Inteligente

Control Inteligente es un conjunto de Técnicas y Esquemas de Control, que basado en los enfoques de la Inteligencia Computacional, tiene como objetivo integrar inteligencia en la Teoría de Control para obtener Sistemas y/o Máquinas Inteligentes.

Ya en 1992, la National Science Foundation (NSF) concluyó que:

> el Control Inteligente debe abarcar tanto la inteligencia como la teoría de control. El control inteligente debe basarse en una tentativa seria de entender y reproducir el fenómeno que siempre hemos llamado "inteligencia", es decir la capacidad de tipo generalizado, flexible y adaptativo que vemos en el cerebro humano. Además, debe estar firmemente arraigado en la teoría de control a la mayor medida posible. Control inteligente es el uso de sistemas de control de propósito general, capaces de aprender con el tiempo como alcanzar objetivos (u optimizar) en entornos no lineales, ruidosos y complejos, cuya dinámica en última instancia debe ser aprendida en tiempo real. Este tipo de control no se puede lograr mediante simples mejoras incrementales sobre los enfoques existentes.

Inteligencia Computacional

Una década antes, en 1983 había publicado el psicólogo Howard Gardner su "Teoría de las inteligencias múltiples", identificando ocho tipos distintos de inteligencia en el ser humano. Desde entonces la Inteligencia

Humana ya fue calificada de compleja, multifacética, subjetiva y relativa a situaciones y habilidades específicas. Esto ha dificultado aun más su reproducción, y los intentos de imitarla en sistemas y/o máquinas han sido numerosos, algunos registrados como enfoques de inteligencia computacional.

Así por ejemplo, para la Sociedad de Inteligencia Computacional de la IEEE Capítulo Chile, IEEE-CIS, (2003):

> La inteligencia computacional es una colección de paradigmas computacionales con inspiración biológica y lingüística, en los cuales se incluyen la teoría, el diseño, la aplicación y el desarrollo de redes neuronales, sistemas conexionistas, algoritmos evolutivos, sistemas difusos y sistemas inteligentes híbridos.

Tabla 1.

Progresos en Inteligencia Computacional

	AI Convencional	Redes Neuronales	Lógica Difusa	Métodos Genéticos
1940s	1947 Cibernéticas	1943 Modelo de Neurona		
1950s	1956 AI	1957 Perceptrón		
1960s	1960 Lenguaje LISP	1960s Adaline, Madeline	1965 Conjuntos Difusos	
1970s	1975 Ingeniería del Conocimiento / Sistemas Expertos	1974 Algoritmo de Retropropagación 1975 Neocognitrón	1974 Controladores Difusos	1970s Algoritmo Genético
1980s	1980s Búsquedas Heurísticas	1980 Mapa auto-organizado 1982 Red Hopfield 1983 Máquina de Boltzmann 1986 Boom del Algoritmo de Retropropagación	1985 Modelamiento Difuso	1980s Modelamiento Inmune de Vida Artificial
1990s		Aplicaciones	1990s Modelos Neuro – Difusos 1991 ANFIS 1994 CANFIS	1990 Programación Genética
2000s			Aplicaciones	Aplicaciones

Fuente: (Cheok, 2002)

En la Tabla 1 se ha registrado el progreso en el desarrollo de cuatro enfoques de Inteligencia Computacional: desde los más antiguos e ingenuos Modelos Conexionistas, que posteriormente darían paso a la teoría de Redes Neuronales, y también desde las primeras interpretaciones de Inteligencia Artificial convencional, hasta llegar a las últimas aplicaciones de Modelos Neuro – Difusos y Programación Genética. A pesar de que no aparecen en la tabla, se debe mencionar que hoy en día se continúa con el desarrollo de nuevos enfoques en áreas tales como: la bioinformática y la bioingeniería, y que en un futuro cercano entregarán nuevas herramientas de control (IEEE-Chile, 2003).

Aunque cada uno de los enfoques se fundamenta en un proceso biológico diferente: mental, evolutivo, o lingüístico, todos tienen como única finalidad el integrar cierto grado de inteligencia a un sistema.

Stuart Russell y Peter Norvig, por ejemplo, en su libro “Artificial Intelligence: A Modern Approach” han publicado en el año 2010 que existen cuatro categorías distintas de inteligencia artificial: sistemas que piensan como humanos, que actúan como humanos, que piensan racionalmente y que actúan racionalmente.

Los sistemas que piensan como humanos intentan imitar el pensamiento humano a través de Redes Neuronales Artificiales (RNA), buscando automatizar actividades realizadas por los procesos del cerebro humano, como:

- Toma de decisiones,
- Resolución de problemas, y
- Aprendizaje

Los sistemas que actúan como humanos intentan imitar el comportamiento humano, este es el caso de la Robótica, donde se estudia cómo lograr que las máquinas hagan tareas, que el ser humano las hace mejor.

Los sistemas que piensan racionalmente tratan de reproducir el pensamiento lógico racional del ser humano a través de cálculos para percibir, razonar y actuar, a estos se los llama sistemas expertos.

Los sistemas que actúan racionalmente, son la categoría más compleja de la inteligencia artificial e intentan emular de una manera idealizada el comportamiento humano, son llamados agentes inteligentes.

Paralelamente, el profesor K. Cheok de la Universidad de Oakland también ha propuesto cinco elementos esenciales que debe tener un sistema para poder ser considerado como agente inteligente:

- Entradas (sensores),
- Salidas (actuadores),
- Memoria (base de datos),
- Reglas (interpretación), y
- Adaptabilidad (capacidad para modificar comportamientos necesarios)

Teoría de Control

Si bien es cierto, en un principio se desarrollaron los esquemas de control inteligente intuitivamente fundamentándolos en procesos biológicos y lingüísticos. Más tarde, cuando los diseños estuvieron definidos, debieron ser entendidos y evaluados en la más profunda matemática, para así conseguir esquemas capaces de controlar sistemas de alta complejidad, para los cuales la Teoría de Control Convencional ya era insuficiente. (National Science Foundation NFS: White, David A.; Sofge, Donald A., 1992)

La base fundamental para la aplicación de esquemas de Control Convencional: ya sea Clásico o Moderno, es el conocimiento de la dinámica del sistema o proceso a controlar, representada mediante Modelos Matemáticos, empleando ecuaciones diferenciales y ecuaciones en diferencias. Sin embargo, estas expresiones matemáticas solamente pueden representar a Sistemas Lineales Invariantes en el tiempo (LTI) y a determinados Sistemas No Lineales. (Babuska, 2001)

Cuando no se dispone del modelo matemático del proceso, o cuando su estructura y sus parámetros cambian impredeciblemente en el tiempo aumentando su complejidad, los sistemas llegan a ser no lineales a un grado tal que las técnicas disponibles en el control convencional son insuficientes e inaplicables (Cotero Ochoa, 2005).

Entonces es necesario emplear controladores diseñados en base a las técnicas introducidas por la teoría del control inteligente, de gran desempeño y capaces: de operar en entornos adversos, de auto gobernarse y de controlar sistemas dinámicos. Además que puedan trabajar con una gran cantidad de datos gracias a los avances en las tecnologías de tratamiento de datos.

En la Figura 1 se ilustra la relación existente entre el control convencional y las técnicas de control inteligente. Mientras el control convencional es *aglutinante* y rígido en su aplicación, los esquemas de control inteligente se caracterizan por ser innovadores y altamente *adaptables* al proceso a controlar.

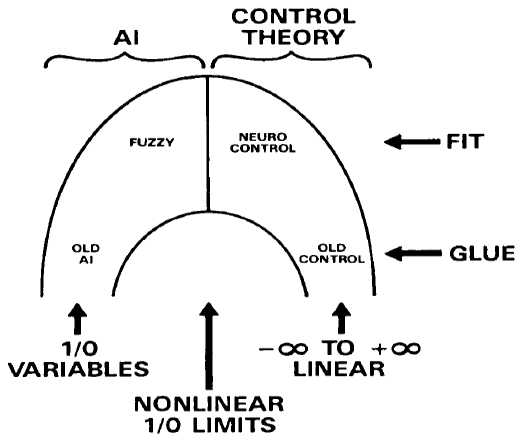


Figura 1. Aspectos de Control Inteligente

Fuente: (National Science Foundation NFS: White, David A.; Sofge, Donald A., 1992)

El Control Neuronal o Neurocontrol -control inteligente basado en redes neuronales artificiales- es la técnica que mejor se ha asentado en la teoría de control, ya que al igual que el control convencional trabaja con variables

continuas sin límite, que están comprendidas entre el infinito negativo y el infinito positivo. Igualmente es innovador pues sus controladores operan correctamente en sistemas lineales y en sistemas no lineales complejos.

El Control Difuso -control inteligente basado en la Lógica Difusa- es el siguiente paso entre la teoría de control y la inteligencia artificial. Permite manipular variables continuas entre cero y uno, o entre cero y más-menos uno. Además es una importante herramienta que solventa los problemas en el razonamiento simbólico formal pues permite adaptar cuantificadores numéricos en inferencias lingüísticas, en base a reglas heurísticas del tipo antecedente - consecuente.

Esquemas de control inteligente más modernos, a pesar de no constar en la Figura 1, comparten la posición central superior con el control difuso. Así por ejemplo, Algoritmos Genéticos, es una técnica que imitando la evolución biológica, ha introducido un grado considerable de inteligencia a la teoría de control. Los algoritmos a semejanza del control difuso también trabajan con variables binarias.

1.5.1. Control Neuronal

Para entender el control neuronal se asume implícitamente la definición de la National Science Foundation NFS, de 1992.

Los sistemas usados en el control inteligente, que son diseñados con redes neuronales artificiales, son conocidos como Controladores Neuronales o Neurocontroladores. Sus esquemas de diseño pueden ser vistos como un problema de optimización no lineal. (Cotero Ochoa, 2005)

1.5.2. Redes Neuronales Artificiales

Las redes neuronales artificiales, intentando imitar los procesos de pensamiento humano, son capaces: de aprender de ejemplos, de realizar

tareas complejas -como la percepción y reconocimiento de patrones-, y de tener tolerancia a fallos en cierta medida. “Pueden ser usadas en modelos no lineales, en sistemas multivariables estáticos y dinámicos, y pueden ser entrenadas mediante el uso de datos de entrada- salida observados en un sistema.” (Babuska, 2001)

Regla de los 100 pasos – Procesamiento Paralelo

Un humano puede reconocer la imagen de una persona u objeto familiar aproximadamente en 100 milisegundos. Si el paso de conmutación neuronal tarda aproximadamente un milisegundo, esta tarea tarda 100 pasos discretos de procesamiento paralelo. En cambio, un ordenador de arquitectura von Neumann no puede hacer nada en 100 pasos de procesamiento secuencial, es decir 100 ciclos de ensamblador. (Kriesel, 2007)

El procesamiento paralelo que realizan las redes biológicas y que es imitado por las artificiales, es otra de las ventajas que tienen sobre las computadoras convencionales empleadas en las técnicas de control clásico.

Entrenamiento de la red

La característica principal de una red neuronal es su capacidad para aprender relaciones funcionales complejas, generalizando a partir de una cantidad limitada de datos de entrenamiento. (Babuska, 2001)

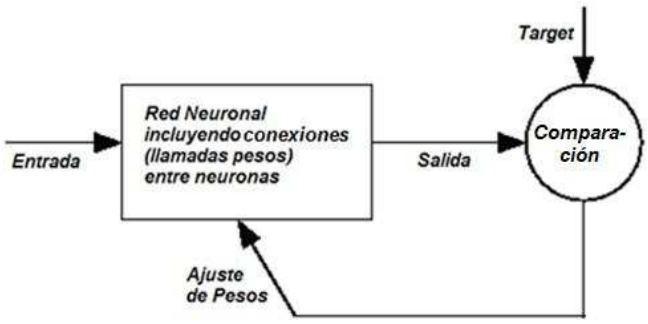


Figura 2. Esquema de entrenamiento de red

Fuente: (Beale, Hagan, & Demuth, 2015)

Por lo general se necesitan muchos datos de entrenamiento para entrenar una red, tal como se ilustra en la Figura 2 denominados pares entrada/target.

Las redes son entrenadas para realizar tareas complejas en diversos campos, incluyendo: el reconocimiento de patrones, la clasificación, la aproximación de funciones, la identificación de sistemas, etc. (Beale, Hagan, & Demuth, 2015)

1.6. MATLAB (Laboratorio de Matrices)

Según MathWorks, desarrollador de MATLAB, (2015):

> Es un lenguaje de alto nivel y un ambiente interactivo para el cálculo numérico, la visualización y la programación. Usando MATLAB, se puede analizar datos, desarrollar algoritmos, y crear modelos y aplicaciones. El lenguaje, las herramientas y las funciones matemáticas incorporadas permiten explorar múltiples enfoques y llegar a una solución más rápida que con lenguajes de programación tradicionales, como C/C++ o Java. Se puede usar MATLAB para una gama de aplicaciones, incluyendo el procesamiento de señales y comunicaciones, el procesamiento de imágenes y videos, los sistemas de control, las pruebas y mediciones, la finanza computacional, y la biología computacional. Más de un millón de ingenieros y científicos en la industria y la academia usan MATLAB, el lenguaje para el cálculo técnico.

1.6.1. Neural Network Toolbox

Es una de las herramientas a disposición en MATLAB, que según la publicación de MathWorks en su sitio web:

> Proporciona funciones y aplicaciones para el modelado de sistemas no lineales complejos, que no son fáciles de modelar con una

ecuación de forma cerrada. Soporta el aprendizaje supervisado con redes: alimentadas hacia adelante, de base radial y dinámicas. También soporta el aprendizaje no supervisado: con mapas de auto organización y capas competitivas. Con el Toolbox se puede diseñar, entrenar, visualizar y simular redes neuronales. Se lo puede usar para aplicaciones como la aproximación de datos, el reconocimiento de patrones, entre otros.

Aplicaciones del Toolbox

Enlistar todas las aplicaciones en las que las redes neuronales proporcionan soluciones resulta complicado, pues están dispersas en una diversidad de campos, como en: la industria aeroespacial, la banca, la medicina, la robótica, el entretenimiento, etc.

En la *robótica* existen cuatro aplicaciones comerciales en las que han tenido un rendimiento superlativo, los sistemas: de *control de trayectoria*, de monta-cargas, de controladores de manipuladores y de visión. (Beale, Hagan, & Demuth, 2015)

1.6.2. MEX Files

Un archivo MEX es una función, creada en MATLAB, para llamar subrutinas programadas en C, C++ o Fortran desde la ventana de comandos, como si se tratase de funciones propias de MATLAB.

El archivo binario MEX está conectado dinámicamente a las subrutinas, que el intérprete de MATLAB las carga y las ejecuta. (MathWorks, 2015) (MathWorks-Help, 2015)

Para soportar esta función, MATLAB tiene características direccionadas al desarrollo y compartimiento de código OPP de programación orientada a objetos.

1.7. Pioneer P3-DX y ARIA

El robot, su microcontrolador, el firmware y los dispositivos integrados, como los sensores ultrasónicos, juntos son llamados plataforma robótica móvil Pioneer P3-DX. (Adept MobileRobots, 2012)

La Pioneer P3-DX, Figura 3, es la plataforma de investigación, educación y experimentación más popular de entre todas las desarrolladas y construidas por MobileRobots Inc – ActivMedia Robotics.

Figura 3. Plataforma móvil Pioneer P3-DX
Fuente: (Adept MobileRobots, 2012)

ARIA es una librería de programación escrita en C++ que proporciona las herramientas necesarias para desarrollar aplicaciones de control para la plataforma robótica. Debe ser vista como un software *cliente/servidor* que brinda un acceso fácil -de bajo o alto nivel- al robot y a sus accesorios.

Los procesos centrales de la plataforma son de tipo *servidor*, es decir están implementados en el firmware ARCOS del sistema operativo Pioneer, que se ejecuta en el microcontrolador.

Estos procesos gestionan las tareas más críticas y sensibles en el tiempo, que son de bajo nivel de control y de operación, incluyendo: el mantenimiento del movimiento solicitado, el reconocimiento del estado de partida y de la posición estimada por odometría, y la adquisición de información de los sensores, entre otras. (Adept MobileRobots, 2012)

El firmware no realiza tareas robóticas de alto nivel. Ese es el trabajo de

un *cliente* inteligente conectado desde una computadora adicional, en la cual se ejecutan aplicaciones de control robótico de alto nivel, como: detección y evasión de obstáculos, y mapeo utilizando los sensores.

Si se desea tener un acceso de bajo nivel se dispone de una clase de Aria –ArRobot- que permite el envió simple de comandos a la plataforma.

Si se desea, por el contrario, tener un control de alto nivel mediante las acciones “Actions” se puede conseguir comportamientos inteligentes. Adicionalmente, Aria proporciona una Interfaz de programación de aplicaciones -API- completa que sirve de base para acceder a otras librerías con capacidades adicionales y específicas.

En consecuencia, las tareas fundamentales que debe realizar Aria son:

- mantener activos y controlados los procesos centrales del servidor,
- soportar la comunicación entre el cliente y el firmware de la plataforma, garantizando la recepción de los envíos desde el cliente por comandos ArRobot o por aplicaciones, y
- soportar todos los dispositivos conectados – a la plataforma y/o al servidor-.

1.7.1. Composición Física de la Pioneer P3-DX

Es un robot móvil compacto, completamente ensamblado con un controlador embebido, dos motores con codificadores de 500 puntos, dos ruedas de diecinueve centímetros, con cuerpo resistente de aluminio, ocho sensores ultrasónicos SONAR direccionados hacia delante y un máximo de tres baterías intercambiables en caliente. Es ideal para el uso en laboratorios internos y en aulas de clase. (Adept MobileRobots, 2011)

En la Figura 4 se muestra un esquema del robot, en vista lateral y superior, sobre el cual se detallan sus dimensiones en milímetros.

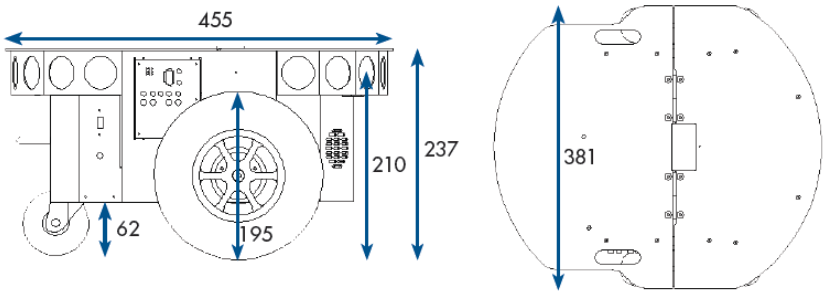


Figura 4. Esquema del Pioneer 3-PX

Fuente: (Adept MobileRobots, 2011)

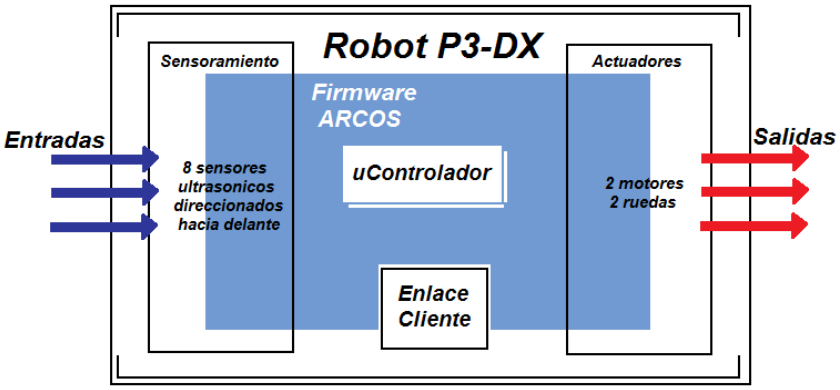


Figura 5. Diagrama del Pioneer P3-DX

En el presente proyecto, el robot se comporta como un agente inteligente (Figura 5). Por lo que debe tener dos elementos físicos esenciales, que de acuerdo al criterio del profesor Ka Cheok de la Universidad de Oakland son:

- elementos de entrada → de sensoramiento (sección 1.7.2), y
- los elementos de salida → de actuación (sección 1.7.3).

1.7.2. Sensores – Anillo Frontal de Sonares

El robot puede saber que sucede en su entorno gracias al equipo de sensamiento integrado: un arreglo frontal de sensores ultrasónicos SONAR de cinco metros de alcance.

Se trata de un conjunto de sensamiento ultrasónico SONAR de ocho transductores de posición fija: uno en cada lado y seis enfocados al frente separados por intervalos de 20 grados (mirar Figura 6).

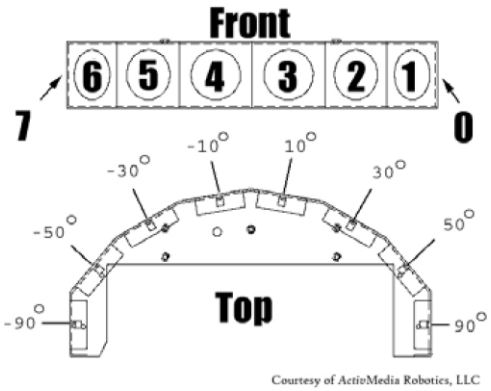


Figura 6. Arreglo de sensor ultrasónicos SONAR P3-DX

Fuente: (MobileRobots - ActivMedia Robotics, 2006)

Cada transductor tiene un ángulo apertura de -15 grados a +15 grados, y proporciona un valor numérico real en cualquier momento. Juntos los transductores alcanzan 180 grados de detección sin áreas obscuras, representados en ocho valores numéricos reales. Todo el arreglo tiene una sensibilidad mínima de diez centímetros y una máxima cercana a cinco metros. (Adept MobileRobots, 2011)

1.7.3. Actuadores – Dos Motores con Codificadores

Los actuadores integrados al robot son dos motores DC reversibles con codificadores de 500 puntos, que están conectados a dos ruedas de accionamiento diferencial. El robot trae además una rueda pivote trasera de equilibrio de menor tamaño.

Tabla 2

Especificaciones de movilidad Pioneer P3-DX

Movimiento	Valor máximo
Radio interno de giro (Figura 7)	26.7 [cm]
Radio de giro producido	0 [cm]
Velocidad máxima transversal (Hacia delante y atrás)	1.2 [m/seg]
Paso transversal máximo	2.5 [cm]
Disparidad trasversal máxima	5 [cm]
Pendiente transversal máxima	25 [grados]
Velocidad rotacional	300 [grados/seg]

Fuente: (Adept MobileRobots, 2011)

Estos actuadores le permiten al robot moverse transversal y rotacionalmente, conforme las especificaciones de la Tabla 2.

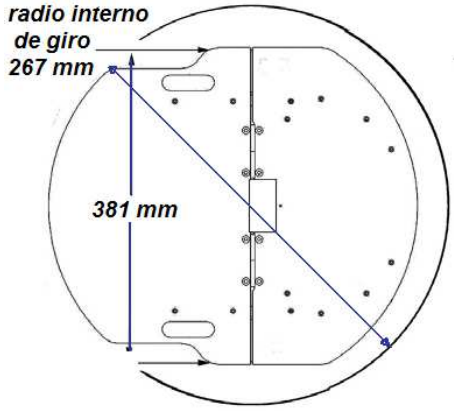


Figura 7. Radio interno de giro - P3-DX
Fuente: (MobileRobots - ActivMedia Robotics, 2006)

El robot es capaz de soportar varios accesorios adicionales, como telémetros laser y pinzas *grippers*, que abrirían un abanico de aplicaciones especializadas. Sin embargo, para el presente proyecto se consideran solo los accesorios integrados.

2. PERCEPTRON MULTICAPA Y CONTROL NEURONAL

2.1. Redes Neuronales Biológicas

Las redes neuronales biológicas han sido la inspiración para el desarrollo de las redes neuronales artificiales. Según Babuska (2001): para obtener una representación funcional de las redes biológicas, estas han sido imitadas de manera muy precisa, compartiendo algunas ventajas que ellas, como organismos biológicos, tienen sobre los sistemas de control estándar.

Primero se debe estudiar la morfología y fisiología de una red biológica para posteriormente poder entender el funcionamiento de su representación artificial.

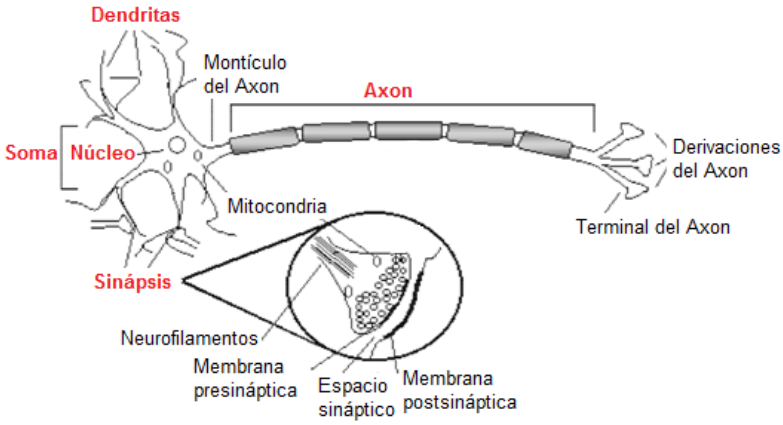


Figura 8. Neurona Biológica

Fuente: (Kriesel, 2007)

Una red neuronal biológica está conformada por varias neuronas comunicadas entre sí mediante pulsos eléctricos de *Activación*. En la Figura

8 se observa la composición morfológica de una de ellas: Las *Dendritas* son las entradas, permiten la recepción de los pulsos de activación de otras neuronas, y están presentes en gran número. El *Soma* o cuerpo celular es el procesador, interpreta los estímulos de entrada recibidos por las dendritas y evalúa la generación de un pulso de salida. El *Núcleo* esta siempre dentro del soma y guarda en su interior el material genético. El *Axón* es el canal de salida, envía el pulso de activación generado por el soma hacia otras neuronas. La *Sinapsis* es el espacio de unión intercelular entre el axón de una neurona y las dendritas de otras, donde se produce el proceso fisiológico de propagación de pulsos eléctricos entre neuronas a través de los neurotransmisores (Bullinaria, 2014).

Todo pulso, que tiene el potencial eléctrico suficiente para superar el umbral crítico de inhibición de una neurona y provocar la generación de un nuevo pulso, es considerado una señal de transmisión sináptica excitadora.

2.2. Redes Neuronales Artificiales

Las redes neuronales artificiales son aproximaciones reales de las redes biológicas, realizadas mediante dispositivos físicos o construcciones matemáticas puras desarrolladas en computadoras convencionales (Bullinaria, 2014). Están compuestas básicamente de entradas, salidas y elementos simples de cálculo llamados neuronas artificiales, que se interconectan entre sí ordenándose en varias capas. Cada una de estas interconexiones tiene asociado un peso de interconexión con información importante para entender la relación entre entradas y salidas (Babuska, 2001).

2.2.1. Neurona de McCulloch - Pitts

En 1943, Warren McCulloch y Walter Pitts publicaron el primer modelo matemático de una neurona artificial: la unidad lógica de umbral McCulloch-Pitts, teniendo gran aceptación. Precisamente este modelo es el inicio

perfecto para entender como han ido evolucionando las representaciones matemáticas de una red neuronal biológica.

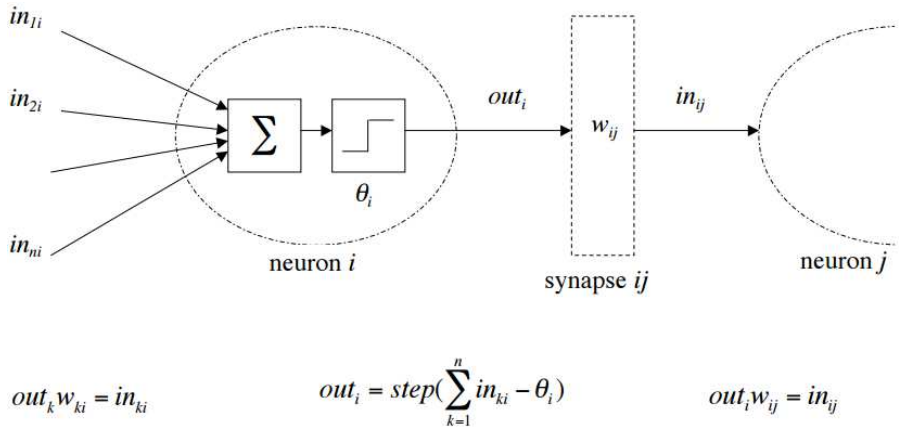


Figura 9. Neurona McCulloch-Pitts

Fuente: (Bullinaria, 2014)

En la Figura 9 se pueden observar dos neuronas McCulloch-Pitts conectadas, la primera señalada con i y la segunda con j. Las in son las entradas, toman valores numéricos reales y representan a las dendritas. La suma ponderada de las entradas menos el valor umbral θ_i es la *regla de propagación* e imita al cuerpo celular Soma. El out_i marca la salida, representado el axón. Ambas neuronas se encuentran conectadas por una fuerza de interconexión llamada *peso* w_{ij} (i por la primera y j por la segunda) que representa a la sinapsis con números reales.

La regla de propagación permite calcular el *potencial postsináptico* de la neurona mediante la suma de sus entradas. Normalmente se considera la importancia de cada una, multiplicándola por el peso que la conecta. Si esta suma ponderada es mayor al valor del umbral de la neurona, estas señales recibidas son consideradas excitadoras pues generan una salida out_i en la primera neurona, su representación matemática es la ecuación central adjunta a la Figura 9.

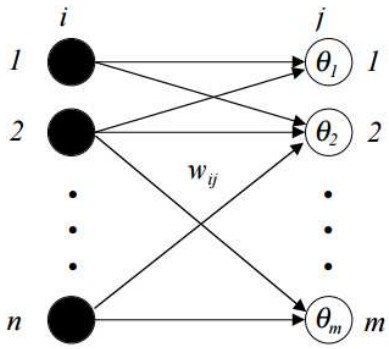
Las ecuaciones de los lados explican que el valor numérico de la entrada de una neurona es el resultado de la multiplicación de la salida de la neurona previa conectada por su respectivo peso. Por ejemplo, el valor de la entrada

de la neurona j se calcula multiplicando el valor de la salida de la neurona i por el peso w_{ij} que las conecta.

2.3. Perceptrón

En 1957 el psicólogo e informático estadounidense Frank Rosenblatt introdujo un primer modelo de red neuronal simple, al que lo llamó Perceptrón.

Rosenblatt juntó varias neuronas McCulloch-Pitts en una configuración de una sola capa de entrada, alimentando hacia adelante a una sola capa de salida. En la Figura 10 se observa un perceptrón, la capa i es considerada de entrada con n posibles entradas, mientras que la capa j es la de salida con un máximo de m neuronas de salida.



$$out_j = \mathrm{sgn}\left(\sum_{i=1}^{n} out_i w_{ij} - \theta_j\right)$$

Figura 10. Perceptrón
Fuente: (Bullinaria, 2014)

Al igual que en la neurona McCulloch-Pitts, también en el perceptrón se calcula el potencial postsináptico. La suma ponderada de las entradas puede ser reemplazada por la suma de los productos de las salidas de la capa anterior por los respectivos pesos de interconexión $(out_i \cdot w_{ij}) = in_j$.

Para decidir si este nuevo *potencial* es excitador, es decir si genera o no una salida out_j, se lo evalúa con la función de activación: *función signo* (definida en la sección 2.4.1). Cuando el potencial es positivo, la función lo evalúa como más uno y genera una activación, en este caso se califica a

dicho potencial de excitador. Por el contrario, cuando es negativo, la función le da un valor menos uno, y es simplemente inhibido. (ecuación adjunta a Figura 10).

La ecuación (1) es una aproximación matemática, en la que se considera al umbral como una entrada inicial del perceptrón ($in_0 = -\theta$). Si a esta entrada se la reemplaza por la multiplicación de la salida de una neurona previa imaginaria por su debido peso ($out_0 \cdot w_0 = -\theta$), y a la salida se la forza a uno ($out_0 = 1$), el umbral puede ser sumado como un peso inicial adicional ($-w_0 = \theta$).

$$out_n = sgn\left(\sum_{i=0}^{n-1} out_{n-1} \cdot w_{n-1}\right) \qquad (1)$$

Cuando se tiene certeza de que se trabaja con un perceptrón simple y de que sus entradas no están conectadas a las salidas de una capa previa, se debe restringir el potencial postsináptico a la suma ponderada de las entradas del perceptrón por sus pesos (2). Además se debe sumar al umbral, que ya es considerado un peso inicial w_o.

$$out = sgn\left(\sum_{k=0}^{n} in_k \cdot w_k\right) \qquad (2)$$

2.3.1. Patrones de Entrenamiento

Al conjunto de datos, que agrupados en un determinado orden se los utiliza en el aprendizaje del perceptrón, se los denomina patrones de entrenamiento. Si el aprendizaje es no supervisado dichos datos corresponden únicamente a las entradas. Pero si el aprendizaje es supervisado, los datos provienen tanto de las entradas, como del objetivo en una suerte de pares, es decir para cada combinación de entradas hay una determinada salida “función objetivo” conocida también como target por su traducción al inglés.

Una de las principales características de las redes neuronales, es la capacidad de aprender de ejemplos y de generalizar comportamientos a partir de estos. Los patrones son los ejemplos del aprendizaje, por lo tanto su correcta generación e interpretación garantiza que el comportamiento del perceptrón sea el esperado (Babuska, 2001).

En la Figura 11 están enlistados los cuatro patrones de entrenamiento que se generan en una compuerta lógica de dos entradas. Tomando como ejemplo el tercer patrón de la compuerta AND enmarcado en rojo se puede analizar: que este patrón permite un entrenamiento supervisado pues está conformado por la combinación de dos entradas (uno en A y cero en B) y por el target de la función, en este caso cero.

Las compuertas lógicas son un caso particular pues sus entradas solo pueden recibir valores binarios: cero o uno, lo que reduce marcadamente la cantidad de combinaciones posibles y por lo tanto de patrones. Normalmente se dispone de una mayor cantidad de valores reales de entrada con su correspondiente salida objetivo, en igualdad o desigualdad lineal.

2.3.2. Separabilidad Lineal

Un perceptrón es capaz de clasificar las compuertas lógicas básicas aproximando su funcionamiento.

Aproximación: las salidas del perceptrón se acercan a los objetivos definidos en cada una de las compuertas, siempre que los valores dados a sus pesos de interconexión y umbrales sean correctos.

Clasificación: la Figura 11 muestra las compuertas lógicas AND, OR y XOR de dos entradas (A y B) y cuatro patrones, cada uno conformado por una combinación distinta de dos entradas y su respectiva función objetivo: uno o cero. Tanto la compuerta AND como la OR son consideradas linealmente separables pues sus patrones pueden ser clasificados por una

recta simple llamada hiperplano, agrupando de un lado a los unos representados con puntos verdes y del otro lado a los ceros en puntos rojos.

		AND	OR	XOR
Entrada A	Entrada B	Salida $A \wedge B$	Salida $A \vee B$	Salida $A \oplus B$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

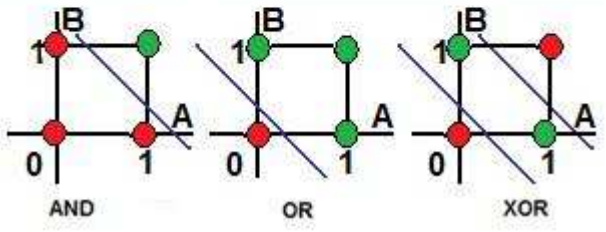


Figura 11. Separabilidad lineal del perceptrón

Por el contrario, la compuerta XOR es considerada linealmente no separable, pues presenta una desigualdad lineal de las salidas con respecto a las entradas. Sus patrones no pueden ser clasificados con una sola recta, sino mínimo con dos (Bullinaria, 2014).

La principal limitación de un perceptrón simple es su incapacidad para resolver problemas linealmente no separables. Si se toma en cuenta que las compuertas lógicas básicas NOT, AND y OR linealmente separables pueden ser aproximadas cada una con un perceptrón, para aproximar las demás compuertas cuya separabilidad ya no es lineal se debe emplear un perceptrón multicapa de arquitectura más compleja, que sea el resultado de juntar varios perceptrones simples.

2.3.3. Límite de decisión - Hiperplano

Se denomina hiperplano al límite de decisión – o superficie de decisión- que clasifica en dos clases a los patrones linealmente separables de un perceptrón simple (Síma, 1998) (Cengiz, 2003). Si los patrones forman un espacio de $n + 1$ dimensiones, entonces el hiperplano es el espacio afín de

dimensión n que divide al espacio primario en dos partes (3).

$$in_1 \cdot w_1 + in_2 \cdot w_2 + \cdots + in_n \cdot w_n + w_0 = 0 \qquad (3)$$

No importa cuántas dimensiones generen los patrones de entrenamiento, siempre existe un hiperplano que los puede clasificar.

Por ejemplo, en la Figura 11 se observa que los patrones de la compuerta lógica OR de dos entradas forman un espacio bidimensional, donde el hiperplano es una recta que divide al plano AB en dos mitades. El hiperplano bidimensional $(in_1 \cdot w_1 + in_2 \cdot w_2 + w_0 = 0)$ debe coincidir con la ecuación de la recta $(y = m \cdot x + b)$. En este caso, las entradas in_n son incógnitas y los pesos w_n son variables manipulables $\left(in_2 = -\frac{w_1}{w_2} \cdot in_1 - \frac{w_0}{w_2}\right)$. De tal manera que la solución analítica del hiperplano sale de las siguientes sustituciones $\left(m = -\frac{w_1}{w_2}\right)$, y $\left(b = -\frac{w_0}{w_2}\right)$.

El cálculo del hiperplano es posible analítica e incluso gráficamente cuando se trabaja con pocas dimensiones. Si éstas aumentan, es necesario emplear Algoritmos de Aprendizaje fundamentados en una matemática más compleja.

2.4. Función de Activación

Es la representación matemática de la tasa o medida, que evalúa si un impulso eléctrico tiene suficiente potencial (un pico de 40 milivoltios) para producir la activación de una sinapsis en una red biológica. A los impulsos que generan una sinapsis se los llama potenciales de activación.

En cada neurona del perceptrón la función de activación genera una salida $out,$ si el potencial postsináptico net percibido llega a ser considerado potencial de activación.

El *potencial postsináptico* es la entrada de la función. Se lo calcula

realizando la suma ponderada de las entradas de la neurona por sus respectivos pesos de interconexión (Gurney, 2007) (Palacios Burgos, 2003).

Los valores de entrada de una función, por lo tanto, pueden oscilar de menos a más infinito $[-\infty, +\infty]$, a diferencia de las salidas que oscilan de cero a uno $[0, +1]$ o de menos uno a más uno $[-1, +1]$, dependiendo de la función seleccionada.

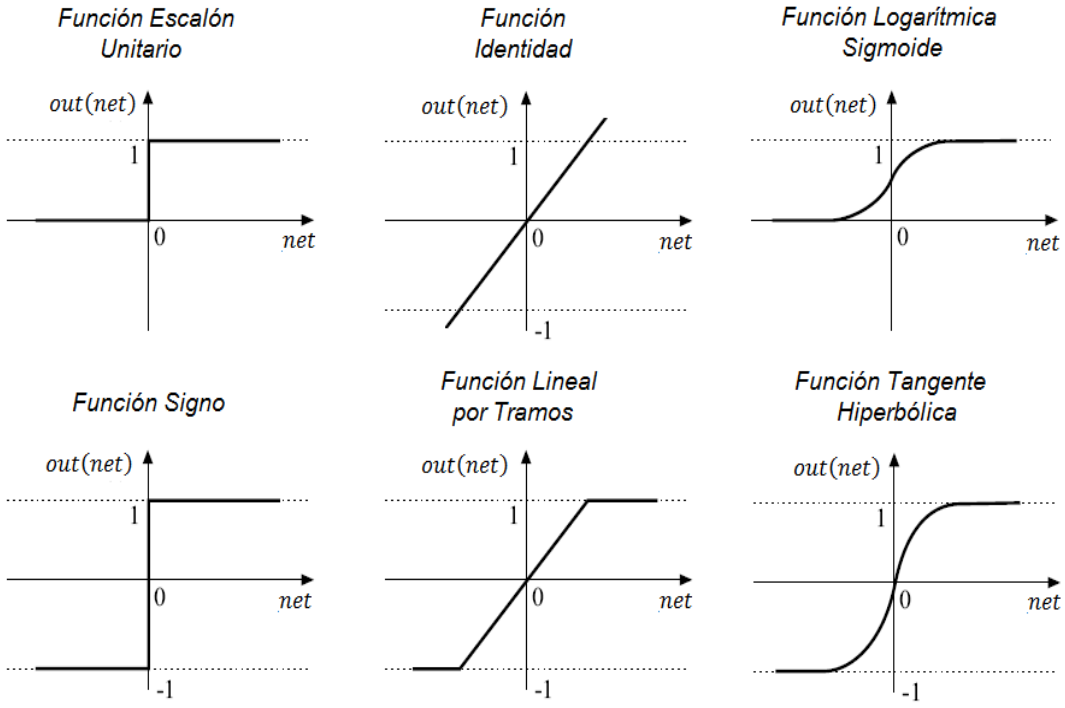


Figura 12. Funciones de Activación

Fuente: (Babuska, 2001)

Las funciones de activación existen de tres tipos fundamentales: umbral, lineal y sigmoidal (Haykin, 2005) (Universidad de Sevilla, 2012). Todas pueden ser utilizadas para determinar las salidas de las neuronas tanto en los perceptrones simples como en los multicapa (Figura 12).

2.4.1. Tipo Umbral

Las funciones umbral son lineales, binarias y discontinuas. Se las conoce como limitadores duros, pues ofrecen solo dos salidas.

Función Escalón Unitario:

Entrega una salida cero ante un potencial postsináptico negativo y uno ante un potencial positivo.

$$out(net) = H(net); \begin{Bmatrix} 0, si\ net < 0 \\ 1, si\ net \geq 0 \end{Bmatrix} \quad (4)$$

Función Signo:

Ante un potencial *net* negativo su salida es menos uno $[-1]$ y ante un positivo es más uno $[+1]$.

$$out(net) = sgn(net); \begin{Bmatrix} -1, si\ net < 0 \\ 0, si\ net = 0 \\ 1, si\ net > 0 \end{Bmatrix} \quad (5)$$

2.4.2. Tipo Lineal

Este tipo de funciones son crecientes, monótonas, lineales y conocidas como de saturación (Universidad de Sevilla, 2012).

Se las usa cuando se necesita saber cómo se incrementa la salida con respecto al potencial postsináptico. Si la función no está acotada, tiene convergencia inestable pues el potencial puede llegar a incrementarse sin límite.

Función Identidad:

Es una función que entrega salidas no acotadas que crecen infinitamente en ambos sentidos.

$$out(net) = net; [-\infty, +\infty] \quad (6)$$

Función Lineal por tramos:

Es definida seccionando en tramos los valores de la variable independiente net dentro del dominio de la función: primera sección $[-\infty, -a]$, segunda sección $[-a, +a]$ y tercera sección $[+a, +\infty]$.

$$out(net) = \begin{Bmatrix} -1, si\ net < -a \\ net, si - a \leq net \leq +a \\ +1, si\ net > +a \end{Bmatrix} \quad (7)$$

2.4.3. Tipo Sigmoidal

Este tipo de funciones son las más empleadas por su comportamiento *"realista"*. Las neuronas biológicas generan salidas continuas, acotadas y no necesariamente lineales, de acuerdo a una tasa o velocidad de reacción. Todas estas características reales han sido recogidas en estas funciones.

Función Logarítmica Sigmoide:

Modelada a partir de la función logística en forma de S -un refinamiento del modelo exponencial-, la función logarítmica sigmoide entrega salidas continuas: partiendo desde valores muy bajos cercanos a cero hasta valores próximos a la asíntota horizontal en uno, con una elevada aceleración en el transcurso.

$$out = sigmoide(net) = \frac{1}{1 + e^{-\alpha \cdot net}} \quad (8)$$

En la ecuación (8) el coeficiente α es la tasa de reacción y representa a la pendiente de la curva sigmoidal: cuando es cercano a cero $(0 \leftarrow \alpha)$ la curva se alarga y tiende a aplanarse, pero cuando aumenta $(\alpha \rightarrow \infty)$ se aproxima a la función umbral.

Esta función además es derivable por lo que puede ser utilizada en el aprendizaje del perceptrón, cuando este se basa en algún algoritmo de

optimización de primer orden o de segundo orden, ya que permite el cálculo de la jacobiano y la matriz hessiana del error total.

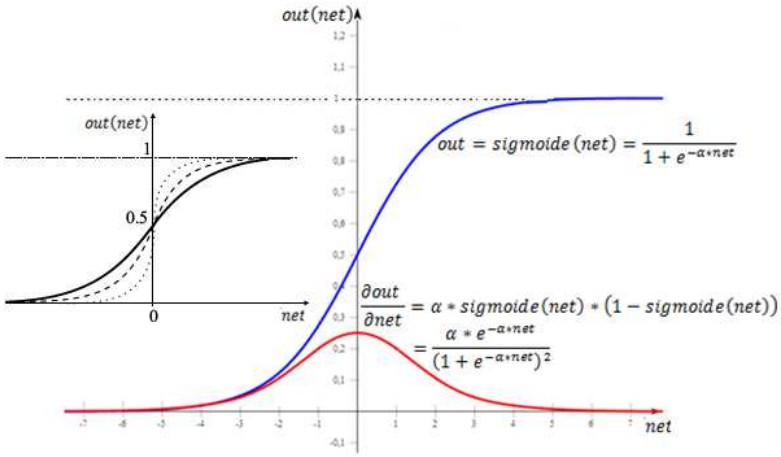


Figura 13. Función Logarítmica Sigmoide

La derivada de la sigmoide ha abierto la puerta a varias funciones alternativas altamente eficientes, que actualmente están siendo usadas en las redes de base radial. Como ejemplo se debe mencionar a la función gausiana $\left(out = A \cdot e^{-B \cdot net^2}\right)$.

Función Tangente Hiperbólica:

También dibuja una curva en forma de S en el rango de menos uno a más uno $[-1, +1]$. Según Andrew Barron, es la función recomendada para calcular las salidas en las capas intermedias de los perceptrones multicapa.

$$out = tanh(net) = \frac{e^{net} - e^{-net}}{e^{net} + e^{-net}} = \frac{1 - e^{-2 \cdot net}}{1 + e^{-2 \cdot net}} \qquad (9)$$

2.5. Aprendizaje

Haykin (2005) describe al aprendizaje como el proceso iterativo mediante el cual una red neuronal artificial aprende sobre su entorno. En cada iteración se realiza un ajuste de los pesos sinápticos, para que la red acumule más conocimiento y mejore su rendimiento en relación a una

función objetivo o a los criterios de control.

Los pesos (incluido el umbral como peso inicial w_o) son los únicos elementos manipulables de entrada, de su adecuada adaptación sistemática depende la correcta aproximación del perceptrón a la función objetivo.

De acuerdo a los datos disponibles para el aprendizaje, este puede ser no supervisado o supervisado.

En el *aprendizaje no supervisado* solo se dispone de los datos de las entradas in_i. La adaptación de los pesos se realiza considerando el criterio de control y las salidas actuales, calculadas a partir de las entradas.

En el *aprendizaje supervisado*, en cambio, el perceptrón cuenta con los valores de las entradas in_i y su respectiva función objetivo *target*, por lo tanto sus pesos se ajustan en función de los datos de entrada y del error calculado entre la función objetivo y sus salidas actuales.

Aprendizaje Supervisado

De tipo adaptativo por corrección del error, el aprendizaje supervisado es un problema de optimización, para el cual se debe tomar en cuenta las dos siguientes definiciones.

Función a minimizar: el error es la función real que se debe minimizar en el aprendizaje. Se lo calcula restando de los valores del objetivo *target* las salidas actuales out del perceptrón.

$$error = target - out \tag{10}$$

Parámetros ajustables de entrada: los únicos elementos que pueden ser modulados son los pesos de interconexión. En la ecuación (11) se ve que son adaptados a partir de su valor previo sumando una pequeña variación, calculada durante la ejecución del algoritmo del aprendizaje. Cada peso

actual $w_{(n)}$ es considerado inicial en la siguiente iteración.

$$w_{(n+1)} = w_{(n)} + \Delta w_{(n)} \tag{11}$$

El objetivo del aprendizaje es adaptar los pesos hasta encontrar aquellos que eliminen el error entre las salidas del perceptrón y los targets especificados. Mientras siga existiendo el *error,* se debe seguir realizando adaptaciones sistemáticas y sucesivas a los pesos de interconexión $\Delta w_{(n)}$.

El tipo de entrenamiento es determinado por la manera en la que ocurren las variaciones de los pesos (Haykin, 2005). En el presente documento se estudian dos algoritmos posibles: *la regla de aprendizaje* o *el descenso de gradiente*, cada uno fundamentado en un principio matemático distinto.

Cuando el aprendizaje supervisado del perceptrón logra anular, o por lo menos minimizar, el error entre las salidas obtenidas y los targets, se puede afirmar que los resultados son positivos y que la función objetivo ha sido correctamente aproximada.

2.5.1. Regla de Aprendizaje

Según Bullinaria (2014), “la regla de aprendizaje de un perceptrón simple se deriva de una consideración de cómo debe desplazarse el hiperplano de decisión a su alrededor, si el problema es linealmente separable”.

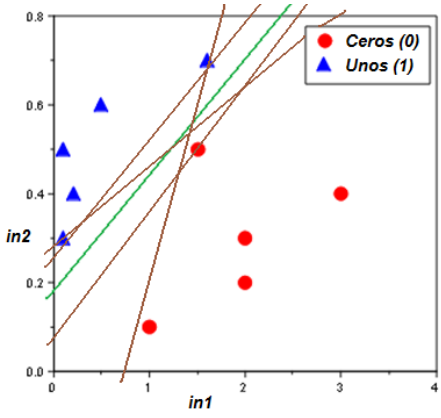


Figura 14. Desplazamientos del hiperplano

El límite de decisión o hiperplano, que es determinado por los pesos de interconexión (3), debe clasificar correctamente a todos los patrones de entrada. Mientras no estén clasificados, se debe continuar haciendo cambios en los pesos $\Delta w_{(n)}$ (Hagan, Demuth, Beale, & De Jesús, 2012)

En la Figura 14 se muestra cómo se dan los desplazamientos de un hiperplano en un espacio bidimensional, mediante la regla de aprendizaje. Las rectas de color café son hiperplanos incorrectos de iteraciones previas, la recta de color verde es el hiperplano final: producto de los desplazamientos anteriores. La ecuación de esta recta hiperplano se resuelve en función de los pesos de interconexión de las dos entradas, cuando dichos pesos varían se generan los desplazamientos de la recta.

Pendiente del hiperplano: $m = -\frac{w_1}{w_2}$

Punto de corte con el eje de ordenadas: $b = -\frac{w_0}{w_2}$

Teoría de Aprendizaje de Hebb

Después de estudiar profundamente el hipocampo (una región del cerebro) en 1949 el neuropsicólogo Donald Hebb publicó la “Teoría de la Asamblea Celular”, en la que propuso una explicación del aprendizaje de las neuronas biológicas.

Según Hebb, la fuerza o peso de la sinapsis se incrementa si las dos neuronas participantes se activan simultánea y sucesivamente, pero si por el contrario se activan sin sincronismo y discontinuamente el peso no aumenta, más bien se reduce hasta la eliminación.

Justamente el algoritmo de la regla de aprendizaje se fundamenta en la teoría hebbiana.

$$\Delta w_{(n)} = \eta \cdot \left(targ_j - out_j\right) \cdot in_i \qquad (12)$$

Con la ecuación (12) se calcula las variaciones de los pesos en un perceptrón, multiplicando tres elementos: η → la tasa de aprendizaje, $(targ_j - out_j)$ → el error calculado con la salida presente de un lado de la sinapsis, e in_i → el valor de la entrada presente al otro lado de la mencionada sinapsis.

La *taza de aprendizaje* es el factor que modula la amplitud de las variaciones, puede tomar un valor real entre cero y uno (Bullinaria, 2014). Por ejemplo, si $\eta = 0.05$ (un valor cercano a cero) se atenúan las variaciones y se necesitan de más iteraciones para observar cambios, por otro lado si $\eta = 0.65$ (un valor cercano a uno) las variaciones son más notorias con el riesgo de generar grandes oscilaciones.

En la teoría hebbiana para que los pesos de la sinapsis se incrementen, se activa específicamente la salida out_j de un lado. Sin embargo en el perceptrón, para evitar que la variación de los pesos tome valores lejanos al target, dicha salida ha sido reemplazada por el error (10).

2.5.2. Descenso de Gradiente

Fundamentado en el algoritmo de optimización de primer orden tiene por objetivo localizar el mínimo global de la función error total en el aprendizaje de un perceptrón: $E = f(w_{(n)})$, produciendo desplazamientos en dirección negativa (o contraria) al gradiente de dicha función desde un punto inicial arbitrario.

En un espacio de $n + 1$ dimensiones se puede representar gráficamente al error total *E* en función de los n respectivos pesos de interconexión de las entradas.

La Figura 15 es un ejemplo de un perceptrón de una entrada, su función error total grafica una curva sobre un plano bidimensional, donde el eje de

las abscisas corresponde al peso y el de las ordenadas al error total.

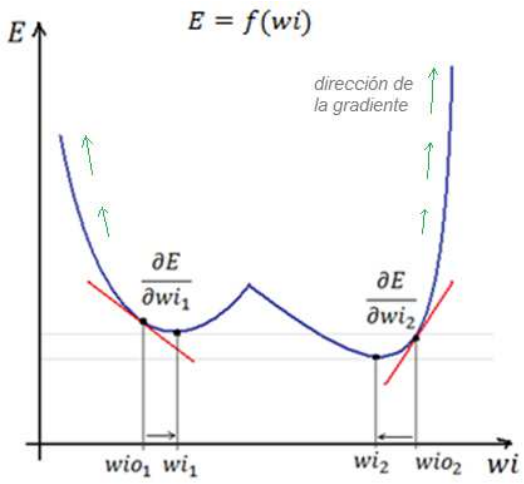


Figura 15. Mínimo local del error - Descenso de gradiente

Si se analizase un perceptrón de dos entradas, la representación gráfica del error total sería un plano bidimensional suspendido en un espacio tridimensional. Así seguirían aumentándose las dimensiones de la representación grafica del error total en relación a la cantidad de pesos disponibles.

Gradiente

Se estudia el gradiente de la función error total *E* con respecto a los pesos: $\nabla E = \nabla f(w_{(n)})$, para entender cómo cambia *E* cuando estos varían.

En un punto inicial arbitrario -dentro del dominio de esta función- se evalúa el vector gradiente para determinar en qué dirección cambia con mayor rapidez (puntos w_{iog} de la Figura 15). La magnitud del vector determina la razón o rapidez del cambio, mientras que la dirección del vector determina la dirección del cambio.

Dirección del gradiente – Puntos Estacionarios

Una propiedad del gradiente afirma que este se anula en los puntos estacionarios -también llamados críticos- de la función, es decir en mínimos,

máximos y puntos de ensilladura. (Larson & Edwards, 2010)

Esto fundamenta el aprendizaje. Una vez determinada la dirección del gradiente, se realizan desplazamientos sucesivos sobre la función error total en dirección contraria (comprobando que su magnitud está descendiendo) hasta encontrar un punto mínimo donde la magnitud sea nula.

Y también lo limita, pues se puede confundir al *mínimo global,* con cualquier local o punto de ensilladura. Por ejemplo en la Figura 15 se evalúa al gradiente en dos puntos iniciales ubicados en regiones distintas de la curva: desde el punto correspondiente a w_{io1} *con* los desplazamientos en dirección contraria al gradiente se localiza al mínimo local w_{i1}, y desde el punto w_{io2} se localiza al local w_{i2} que es el mínimo global de toda la curva.

El aprendizaje por descenso de gradiente logra hallar el *mínimo global* de la función error total *E*, si y solo si el punto de inicio elegido w_{io} está en sus cercanías. De no ser así, se debe empezar nuevamente el aprendizaje con valores distintos en los pesos iniciales w_{io}.

Magnitud del gradiente – Derivada Parcial de 1er Orden

Se puede aproximar el valor de la magnitud del gradiente calculando la derivada parcial de primer orden de la función error total en un punto dado. En una función real de varias variables independientes, como el error total *E* con los pesos, la derivada parcial de primer orden calcula la rapidez con la que *E* cambia según cambie uno de sus pesos de interconexión mientras los otros permanecen constantes. (Weisstein, 2015)

$$\nabla E\left(w_{(n)}\right) = \frac{\partial E}{\partial w_{(n)}} \tag{13}$$

La derivada de la función error total en un punto es igual a la pendiente de la línea tangente a la curva en ese punto. Por ejemplo en la Figura 15 están señalados dos puntos: w_{io1} y w_{io2}, que tienen dibujadas sus

respectivas rectas tangentes en color rojo. Para saber cuál es su pendiente, se debe calcular la función tangente del ángulo entre cada recta tangente y el eje positivo de las abscisas.

Cuando la recta tangente es completamente horizontal y paralela al eje de las abscisas, el valor de la pendiente (es decir de la derivada y del gradiente) es igual a cero. La tangente entonces esta graficada en un punto estacionario de la curva, ya sea este mínimo o máximo o de ensilladura. En el ejemplo las tangentes graficadas en plomo corresponden a puntos estacionarios: en w_{i1} un mínimo local y en w_{i2} el mínimo global.

Si el módulo del gradiente es positivo $\left(\frac{\partial E}{\partial wi} > 0\right)$ y el error total aumenta cuando el peso de interconexión también aumenta, se debe disminuir el valor del peso, caso w_{io2} en la Figura 15. Si por el contrario el módulo del gradiente es negativo $\left(\frac{\partial E}{\partial wi} < 0\right)$ y el error total disminuye cuando se aumenta el peso, se debe entonces seguir aumentando el peso, caso w_{io1} en la Figura 15).

Si finalmente el módulo del gradiente es nulo $\left(\frac{\partial E}{\partial wi} = 0\right)$ se ha encontrado un punto estacionario, no se debe variar el peso pero sí se debe verificar que se trate de un mínimo global.

Error total – error cuadrático medio

El error total se calcula realizando una suma del *error* (10) de cada salida y de cada patrón de entrenamiento del perceptrón.

El *error* puede ser: negativo si el valor de la salida actual es mayor que el objetivo, y positivo si el objetivo es mayor. En ambos casos se genera un error, el problema aquí es que un negativo no suma al error total, por el contrario lo resta. La forma de evitar esto es trabajar con la cantidad puramente positiva obtenida del cuadrado de la diferencia. (Gurney, 2007)

$$E = (error)^2 = (targ - out)^2 \quad (14)$$

Luego se suma el *error total* de todas las salidas actuales (sumatoria subíndice k) de todos los patrones de entrenamiento (sumatoria subíndice p). (Bullinaria, 2014)

$$E(w) = \frac{1}{2}\sum_{p}\sum_{k}(targ \quad - out_k)^2 \quad (15)$$

Cálculo de la Variación

Después de haber aclarado todas las definiciones previas se debe analizar la expresión matemática (16) que determina las variaciones de cada peso de interconexión calculando la derivada parcial de primer orden del error total *E* con respecto a cada peso.

$$\Delta w_{(n)} = -\eta \frac{\partial E}{\partial w_{(n)}} \quad (16)$$

Al igual que en la regla de aprendizaje, en el descenso de gradiente las variaciones también son atenuadas por la tasa de aprendizaje η pero en valor negativo.

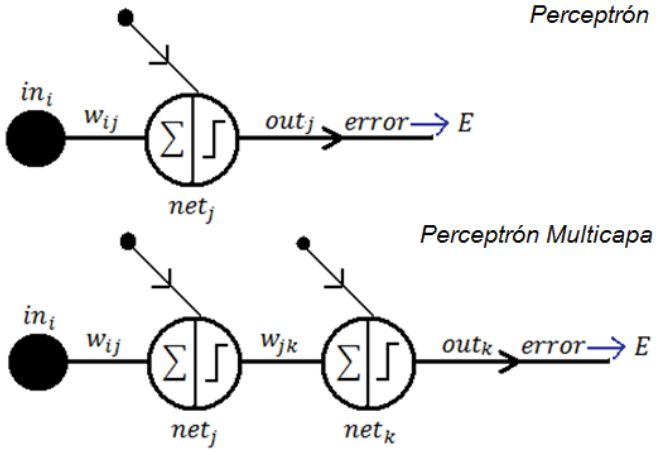


Figura 16. Esquema de un Perceptrón

Para entender el cálculo de la derivada parcial de primer orden, se toma de ejemplo a un *perceptrón simple* (Figura 16), en el que se representan cada una de sus variables y la relación que existe entre ellas.

La derivada es descompuesta mediante la *Regla de la cadena* considerando cada una de las relaciones detalladas en el perceptrón simple.

$$\frac{\partial E}{\partial wij} = \frac{\partial E}{\partial error} \cdot \frac{\partial error}{\partial outj} \cdot \frac{\partial outj}{\partial netj} \cdot \frac{\partial netj}{\partial wij} \tag{17}$$

El error total *E* se calcula a partir del *error* (15). La ecuación (18) determina su derivada.

$$\frac{\partial E}{\partial error} = \frac{\partial \left(\frac{1}{2} error^2\right)}{\partial error} = error = \left(targ - out_j\right) \tag{18}$$

El *error* es la diferencia entre el objetivo target y la salida actual out_j del perceptrón. Al derivar el *error* respecto de la variable dependiente *outj* se obtiene como resultado menos uno.

$$\frac{\partial error}{\partial outj} = \frac{\partial (targ - out_j)}{\partial outj} = -1 \tag{19}$$

Una salida actual se genera cuando el potencial postsináptico de la neurona net_j es evaluado por una función de activación.

La función signo no es derivable, por lo tanto no se la puede utilizar en este aprendizaje.

Para calcular la derivada de out_j con respecto net_j (21), se emplea la función logarítmica sigmoide derivable (Figura 13). Considerar al potencial postsináptico como la variable independiente y a la salida como la función.

$$outj = sigmoide(netj) = \frac{1}{1 + e^{-\alpha \cdot netj}} \tag{20}$$

$$\frac{\partial outj}{\partial netj} = \alpha \cdot sigmoide(netj) \cdot \big(1 - sigmoide(netj)\big) \ ;$$

$$\frac{\partial outj}{\partial netj} = \alpha \cdot out_j \cdot \big(1 - out_j\big) \qquad (21)$$

El potencial postsináptico net_j es la suma ponderada de la entrada in_i por su respectivo peso. La ecuación (22) permite obtener su derivada.

$$\frac{\partial netj}{\partial wij} = \frac{\partial}{\partial wij}\big(in_o \cdot w_o + in_1 \cdot w_1 + \cdots + w_{ij} \cdot in_i\big) = in_i \qquad (22)$$

Una vez resueltas cada una de las derivadas parciales en la regla de la cadena, se realiza las debidas sustituciones.

$$\frac{\partial E}{\partial wij} = \big(targ - out_j\big) \cdot (-1) \cdot \Big(\alpha \cdot out_j \cdot \big(1 - out_j\big)\Big) \cdot in_i \qquad (23)$$

Con la ecuación (24) se calcula finalmente la variación de los pesos de interconexión con el algoritmo del descenso de la gradiente. A la tasa de reacción de la función sigmoide se da un valor típico igual a cuatro $(\alpha = 4)$.

$$\Delta w_{(n)} = -\eta \cdot \big(targ - out_j\big) \cdot (-1) \cdot \Big(\alpha \cdot out_j \cdot \big(1 - out_j\big)\Big) \cdot in_i \qquad (24)$$

En un *perceptrón multicapa* el aprendizaje por descenso de gradiente se torna complejo. Las variables dentro del perceptrón aumentan como se puede observar en la Figura 16, por lo que la regla de la cadena debe considerar más derivadas parciales entre variables relacionadas.

Además el gradiente debe ser determinado considerando todas las dimensiones (es decir considerando todos los pesos), haciéndose necesario el cálculo del gradiente de primer orden del error total (sección 2.8.1 – Algoritmo de retropropagación del error).

2.6. Arquitectura

Matich (2001) afirma que “la arquitectura de una red neuronal consiste en la organización y disposición de las neuronas en la misma”. A las neuronas se las debe visualizar arregladas en Capas. Las neuronas de una misma capa tienen el mismo comportamiento, pues comparten la misma función de activación y el mismo diseño de conexión a otras capas (Fausett, 1994). Las capas pueden estar conectadas: o alimentando hacia delante o recurrentemente.

Las *redes alimentadas hacia delante* no tienen ninguna conexión en retroalimentación. Pueden ser redes simples -como el perceptrón- o pueden ser redes multicapa, que además de la capa de entradas y de la capa de salida tienen otras capas intermedias escondidas u ocultas.

Cada capa alimenta a la siguiente, por ejemplo a la izquierda de la Figura 17 la red alimentada hacia delante tiene dos capas escondidas. La primera (de cuatro neuronas) está alimentada por la capa de entrada, la segunda (de tres neuronas) alimentada por la primera capa escondida y la de salida (de dos neuronas) alimentada por la segunda capa escondida.

Las *redes recurrentes* deben tener conexiones en retroalimentación, sin importar si tienen una sola capa o varias. A la derecha de la Figura 17 se observa la red recurrente de una sola capa que divulgó John Hopfield en 1982.

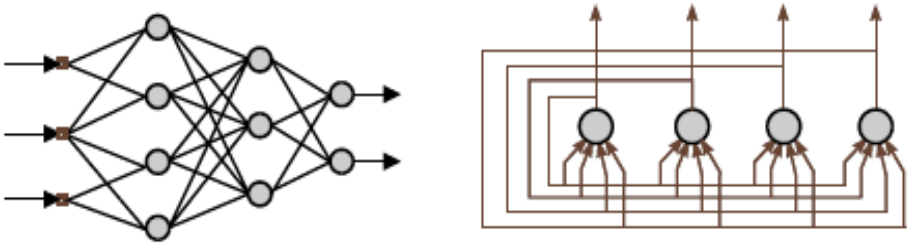


Figura 17. Arquitecturas de red

Fuente: (Babuska, 2001)

2.7. Perceptrón Multicapa

Se tuvo que esperar hasta 1986 para que el psicólogo David Rumelhart continúe con el estudio de las redes neuronales artificiales. Fue el primero en diseñar una red multicapa cuando formulaba el algoritmo del entrenamiento por retropropagación del error.

Para clasificar correctamente patrones de entrenamiento linealmente no separables, un perceptrón simple debe ser modificado cambiando su arquitectura. (Zurada, 1992)

Un perceptrón multicapa es una red estática de arquitectura alimentada hacia delante con una o varias capas escondidas entre la capa de entrada y la de salida. Cada capa está compuesta por perceptrones simples, de características lineales. (Zurada, 1992)

Su atributo más importante es la capacidad para aprender asignaciones entrada/salida de cualquier complejidad. Logrando que aparte de las tareas de clasificación, pueda –entre otras tareas- aproximar cualquier función y controlar sistemas dinámicos. (Zurada, 1992)

Sistemas Dinámicos

Para poder resolver problemas de identificación y optimización de sistemas dinámicos se debe darle características dinámicas a esta red estática, generando una conexión en retroalimentación desde sus salidas hasta sus entradas (Babuska, 2001). Esta conexión no debe ser confundida con la arquitectura recurrente.

Por ejemplo, la Figura 18 es una simulación desarrollada en Simulink de un perceptrón multicapa -de tres entradas y dos salidas- representando a un sistema dinámico de primer orden con dos variables de estado. Cada una de estas variables es representada mediante una salida del perceptrón

retroalimentada hasta una entrada con un operador retardo $\left(\frac{1}{z} = z^{-1}\right)$. La entrada restante recibe a la señal de control $u(k)$ del sistema dinámico.

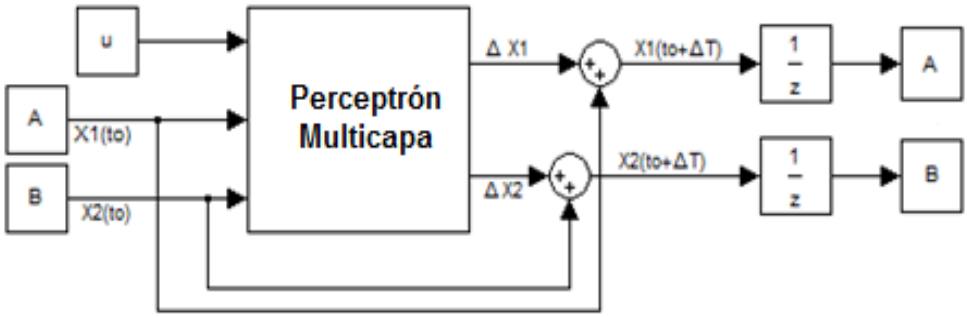


Figura 18. Diagrama de un sistema dinámico de 1er orden

Con el retardo se logra que las variables de estado $X1$ y $X2$ sean retroalimentadas al estado siguiente. Las variables del estado actual *k*: $X1(k) = X1(to + \Delta t)$ y $X2(k) = X2(to + \Delta t)$, son consideradas condiciones iniciales de las variables del estado siguiente: $X1(k+1) = X1(to)$ y $X2(k+1) = X2(to)$.

De este modo, las variables del estado siguiente se calculan dentro del perceptrón multicapa, representado por el subíndice pm, en función de sus condiciones iniciales y de la señal de control en el estado actual: $X1(k+1) = f_{pm}(X1(k), u(k))$ y $X2(k+1) = f_{pm}(X2(k), u(k))$.

2.7.1. Criterios de Aproximación

El segundo teorema de Cybenko (1989) dice: "una red neuronal alimentada hacia adelante, con al menos una capa oculta con funciones de activación sigmoidal, puede aproximar cualquier función no lineal continua $R^p \rightarrow R^n$ correctamente con un conjunto compacto de suficientes neuronas ocultas habilitadas".

De acuerdo a este teorema, un perceptrón multicapa puede aproximar con gran precisión cualquier función, siempre que se consideren los siguientes criterios de diseño:

Número de Capas Ocultas: con una sola capa oculta se obtienen resultados teóricamente satisfactorios. (Babuska, 2001)

Se debe considerar, que -si bien es cierto- con varias capas se pueden obtener leves mejorías en la aproximación, el entrenamiento –en cambio- se hace lento y ocasionalmente hasta inconcluso.

Número de Neuronas en la Capa Oculta: es el criterio de diseño más influyente. Para conseguir una buena aproximación se debe elegir el número correcto de neuronas.

Aunque se lo puede determinar con un método empírico de prueba-error, se recomienda calcularlo en función del error tota*l E*, considerando el criterio del doctor Andrew Barron:

> Una red neuronal alimentada hacia delante con una capa oculta con funciones de activación sigmoidal puede conseguir un error cuadrado integrado (para funciones suaves) del orden
>
> $$E = \mathcal{O}\left(\frac{1}{h}\right) \tag{25}$$
>
> independientemente de la dimensión en el espacio de entrada, el coeficiente h denota el número de neuronas ocultas.

Por ejemplo: con ocho neuronas en la capa oculta ($h = 8$) el *error total* se aproxima a $E = \mathcal{O}\left(\frac{1}{8}\right) \cong 0.125$, pero con 24 neuronas $(h = 24)$ se aproxima a $E = \mathcal{O}\left(\frac{1}{24}\right) \cong 0.0416$.

Pocas neuronas generan una aproximación mala, mientras que demasiadas, sobreentrenan al perceptrón dejando poco espacio a la generalización de datos.

Funciones de Activación: Özkan & Sunar Erbek (2003) y Suttisinthong &

Seewirote (2014) en sus trabajos científicos han concluidos que:
La elección de la función de activación, en las capas escondidas y en la capa de salida, puede hacer variar el rendimiento de la red multicapa. Las funciones sigmoidales -logarítmica y tangente hiperbólica- y lineales han sido usadas efectivamente con los perceptrones multicapa en varios propósitos. Si se trabaja con un perceptrón de una sola capa oculta, la combinación de funciones: tangente hiperbólica en la capa oculta y lineal en la de salida da la más alta precisión en la aproximación. En el caso de un perceptrón con dos capas ocultas, ambas trabajan con la misma función tangente hiperbólica.

Tabla 3.

Combinación de Funciones de Activación

Capa	Primera oculta	Segunda oculta	De salida
Perceptrón de una capa oculta	Tangente Hiperbólica	-	Lineal
Perceptrón de dos capas ocultas	Tangente Hiperbólica	Tangente Hiperbólica	Lineal

Fuente: (Özkan & Sunar Erbek, 2003) (Suttisinthong & Seewirote, 2014)

2.7.2. Aprendizaje del perceptrón multicapa

En el aprendizaje se distinguen dos fases: el cálculo hacia delante de las salidas y la adaptación de los pesos hacia atrás. “Este cálculo hacia atrás es llamado retropropagación del error”. (Babuska, 2001)

Cálculo hacia delante de las salidas

Las salidas siempre son calculadas ordenadamente de atrás hacia delante, es decir desde la primera capa oculta hasta la capa de salida. En la capa de entrada no hay neuronas, solo se agrupan todas las entradas in_i , conectadas a la primera capa escondida a través de los respectivos pesos de interconexión w_{ij}^{1h} (Figura 19).

Para no confundir las variables, estas traen un superíndice adjunto, h para las capas escondidas y o para la capa de salida. Por ejemplo, w_{ij}^{1h} y

out_j^{1h} es la notación de los pesos y las salidas de la primera capa escondida.

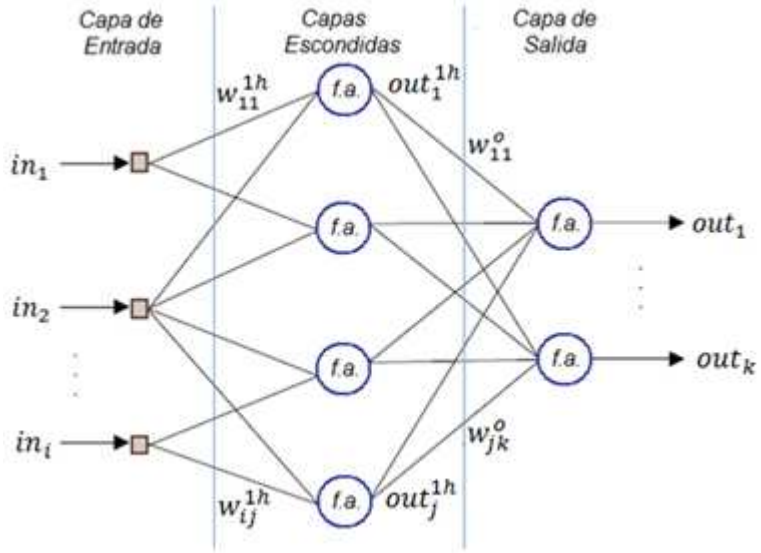


Figura 19. Perceptrón multicapa

Entre la primera capa escondida ($l = 1$, de j posibles neuronas) y la capa de salida (o, de k posibles neuronas) puede haber más capas escondidas, numeradas con el superíndice l.

Con $f.a.$ se señala la función de activación de cada neurona del perceptrón. Todas las neuronas de una capa comparten una función de activación, pudiendo ser ésta desde la más básica de tipo umbral hasta la más sofisticada de tipo sigmoidal.

Las salidas de la primera capa oculta out_j^{1h} son las primeras en ser calculadas a partir de las entradas del perceptrón in_i .

$$out_j^{1h} = f.a.\left(\sum_{m=0}^{i} w_{ij}^{1h} \cdot in_i\right) \tag{26}$$

Si hay una segunda capa oculta, sus salidas se calculan a partir de las salidas de la primera oculta. Si hay una tercera, sus salidas se calculan entonces a partir de las de la segunda, así sucesivamente hacia delante hasta llegar a la capa de salida.

La ecuación (27) permite determinar las salidas de la capa de salida a partir de las salidas de la última capa escondida.

$$out_k = f.a.\left(\sum_{r=0}^{j} w_{jk}^{o} \cdot out_j^{lh}\right) \quad (27)$$

Variación de pesos hacia atrás

Una vez obtenidas las salidas del perceptrón multicapa -es decir, las salidas de la capa de salida- se debe determinar el *error*, comparándolas con las salidas de la función objetivo.

Para minimizar este error se ajustan ordenadamente los pesos hacia atrás, empezando por los de la capa de salida w_{jk}^{o} y terminando con los de la primera capa escondidaw_{ij}^{1h}, (sección 2.8.1).

2.8. Entrenamiento del perceptrón multicapa

El aprendizaje y el entrenamiento por retropropagación del error se complementan. Cuando el aprendizaje ha terminado el cálculo hacia delante de las salidas actuales, el entrenamiento empieza a variar los pesos hacia atrás para lograr minimizar el *error*.

Para el entrenamiento se dispone de algunos algoritmos, fundamentados en tres principios matemáticos: la gradiente de la función, la curvatura de la función y las gradientes conjugadas.

Gradiente de primer orden: el algoritmo de retropropagación del error se basa en este principio. *Curvatura (gradiente de segundo orden):* Newton y Levenberg-Marquardt se basan en este principio. Muchas veces estos algoritmos son considerados más efectivos que los de primer orden.

2.8.1. Algoritmo de Retropropagación del error

El entrenamiento por retropropagación del error debe ser considerado un problema de optimización no lineal de primer orden, que busca minimizar el *error* (10), usando el cálculo del gradiente. (Cotero Ochoa, 2005) (Babuska, 2001)

Mientras en el aprendizaje por descenso de gradiente (sección 2.5.2) se calcula la variación de los pesos mediante el gradiente de primer orden de la función *error total E* respecto de un peso a la vez. En la retropropagación se debe calcular la variación mediante dicho gradiente pero con respecto a todos los pesos en un mismo instante.

$$\Delta w_{(n)} = -\eta \cdot \nabla J\left(w_{(n)}\right) \tag{28}$$

Por cada peso hay una derivada parcial de primer orden que calcula el gradiente del error total. La matriz que agrupa a todas estas derivadas se la llama *Jacobiana*.

$$\nabla J\left(w_{(n)}\right) = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_q}\right]^T \tag{29}$$

Variación de pesos hacia atrás

Cuando el error ya ha sido determinado (a partir de las salidas calculadas hacia delante), debe ser minimizado variando los pesos. La variación se realiza ordenadamente de atrás hacia delante, empezando con los pesos de la capa de salida.

Capa de salida

Los desplazamientos sobre la curva del error total en dirección contraria al gradiente deben ser calculados únicamente en función de los pesos que conectan a esta capa w_{jk}^{o} (Babuska, 2001). A semejanza del aprendizaje por

descenso de gradiente del perceptrón simple, el cálculo del jacobiano es soportado por la regla de la cadena. (Nomenclatura usada en cálculo, de acuerdo a la Figura 19).

$$\frac{\partial \nabla J}{\partial w_{jk}^{o}} = \frac{\partial \nabla J}{\partial error} \cdot \frac{\partial error}{\partial out_{k}^{o}} \cdot \frac{\partial out_{k}^{o}}{\partial w_{jk}^{o}} \tag{30}$$

La derivada del *error total* respecto del *error* (18) y la derivada del *error* respecto de las salidas del perceptrón (19) ya son conocidas.

Normalmente la función de activación en esta capa es de tipo lineal, por lo que las salidas son idénticas al potencial postsináptico.

Las salidas de la capa oculta anterior inmediata son consideradas las entradas de la actual.

$$out_{k}^{o} = \sum_{j} \left(out_{j}^{lh} \cdot w_{jk}^{o} \right) \tag{31}$$

La ecuación (32) permite determinar la derivada de la salida de la capa de salida respecto de los pesos que la alimentan.

$$\frac{\partial out_{k}^{o}}{\partial w_{jk}^{o}} = out_{j}^{lh} \tag{32}$$

La variación de los pesos de la capa de salida (33) siempre debe ser la primera en ser determinada. Se la calcula tomando en cuenta el objetivo *target*, su salida out_{k}^{o} y la salida de la capa escondida anterior out_{j}^{lh}.

$$w_{jk_{(n+1)}}^{o} = w_{jk_{(n)}}^{o} + \Delta w_{jk_{(n)}}^{o} \; ; \quad si: \Delta w_{jk_{(n)}}^{o} = -\eta \frac{\partial \nabla J}{\partial w_{jk}^{o}} \; ;$$

$$\frac{\partial \nabla J}{\partial w_{jk}^{o}} = -out_{j}^{lh} \cdot (targ - out_{k}^{o}) \tag{33}$$

Primera capa oculta

Después de haber obtenido la variación de los pesos de la capa de salida, se debe continuar hacia atrás con la siguiente capa. Normalmente un perceptrón multicapa tiene una sola capa escondida.

En este caso, los desplazamientos sobre el error total son calculados en función de los pesos que conectan las entradas w_{ij}^{1h}.

$$\frac{\partial \nabla J}{\partial w_{ij}^{1h}} = \frac{\partial \nabla J}{\partial out_j^{1h}} \cdot \frac{\partial out_j^{1h}}{\partial net_j^{1h}} \cdot \frac{\partial net_j^{1h}}{\partial w_{ij}^{1h}} \tag{34}$$

El error total se pondera respecto de la salida actual out_j^{1h} de cada una de las neuronas de la capa, multiplicándolo por su respectivo peso de interconexión.

$$\frac{\partial \nabla J}{\partial out_j^{1h}} = \sum_j (targ - out_k^o) \cdot w_{jk}^o \tag{35}$$

La función de activación en la capa oculta es la logarítmica sigmoide. La derivada de la salida respecto del potencial postsináptico net_j^{1h} de esta función (21) y la derivada del potencial respecto de los pesos (22) ya han sido calculadas.

La ecuación (36) de la variación de los pesos de la primera capa escondida se agranda dependiendo del número de neuronas. A más neuronas hay más pesos w_{jk}^o conectados a la capa de salida.

$$w_{ij}^{1h}{}_{(n+1)} = w_{ij}^{1h}{}_{(n)} + \Delta w_{ij}^{1h}{}_{(n)}; \quad si: \Delta w_{ij}^{1h}{}_{(n)} = -\eta \frac{\partial \nabla J}{\partial out_j^{1h}} \quad ;$$

$$\frac{\partial \nabla J}{\partial out_j^{1h}} = -in_i \cdot \left(\alpha \cdot out_j^{1h}\left(1 - out_j^{1h}\right)\right) \cdot \sum_j (targ - out_k^o) \cdot w_{jk}^o \tag{36}$$

Épocas

Babuska (2001) llama época a cada nueva presentación de un nuevo conjunto de datos.

En la primera época se escogen valores arbitrarios para los pesos iniciales de interconexión. Con estos se calcula las salidas actuales del perceptrón (de capa en capa hacia delante) y el error total *E*. Una vez calculado *E* se empiezan a variar los pesos (hacia atrás) con el objetivo de minimizarlo. Los nuevos pesos variados son los datos de la siguiente época. Se deben generar tantas épocas como sean necesarias hasta que el error total es lo suficientemente bajo.

2.8.2. Entrenamiento de Segundo Orden - Curvatura

Este entrenamiento debe ser entendido como un problema de optimización no lineal de segundo orden. Mientras los algoritmos de primer orden se centran en el cálculo del gradiente de la función error total (es decir, la derivada parcial de primer orden del error total con respecto de los pesos), los métodos de segundo orden se centran en la curvatura (la derivada parcial de segundo orden).

A la función *error total* $E = f(w_1, w_2, \dots, w_q)$ se le calcula las derivadas parciales de primer orden respecto de los pesos de interconexión (29). Si las derivadas conseguidas siguen estando en función de los pesos, éstas pueden ser nuevamente derivadas.

Cálculo de las Variaciones

Todas las derivadas parciales de segundo orden son agrupadas en la Matriz Hessiana $H(w_{(n)})$.

$$H\left(w_{(n)}\right) = \left[\frac{\partial^2 E}{\partial \mathrm{w}_1^2}, \frac{\partial^2 E}{\partial \mathrm{w}_2^2}, \dots, \frac{\partial^2 E}{\partial \mathrm{w}_\mathrm{q}^2}\right] \ ;$$

$$H(w_{(n)}) = [f_{w1w1}(w_1, \dots, w_q), \dots, f_{wqwq}(w_1, \dots, w_q)] \quad (37)$$

Para calcular las variaciones de los pesos $\Delta w_{(n)}$ en un método de segundo orden, se debe multiplicar el jacobiano de la función error total por su respectiva matriz hessiana con potencia negativa. (Babuska, 2001)

$$w_{(n+1)} = w_{(n)} + \Delta w_{(n)} \;\; ; \;\; donde$$
$$\Delta w_{(n)} = -H^{-1}(w_{(n)}) \cdot \nabla J(w_{(n)}) \quad (38)$$

A la ecuación (38) se la conoce como el paso básico del método de Newton, pilar fundamental en la definición del resto de algoritmos de entrenamiento rápido. (Beale, Hagan, & Demuth, Reference Neural Network Toolbox MATLAB R2015a, 2015)

Curvatura del error total E – Matriz Hessiana

Las derivadas parciales de segundo orden definen la curvatura de la función, y la curvatura define el carácter de sus puntos estacionarios.

Mínimo Local

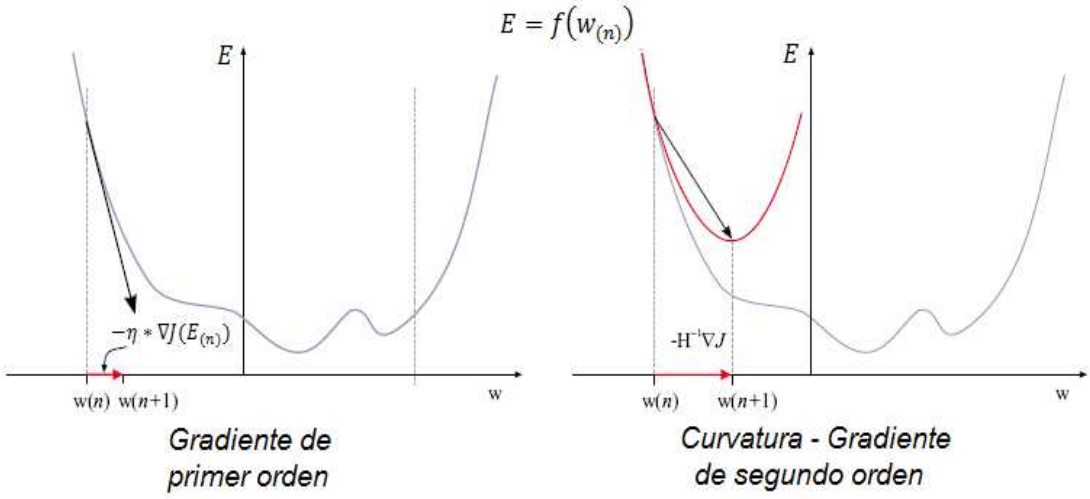


Figura 20. Entrenamiento de segundo orden

Fuente: (Babuska, 2001)

Si en un punto inicial arbitrario $w_{(n)}$, mostrado en la Figura 20, el determinante de la matriz hessiana es positivo $\left|H\left(w_{(n)}\right)\right| > 0$, entonces la función error total tiene forma convexa en la sección próxima. Cualquier punto cerca, que tenga un gradiente cero $\nabla J\left(w_{(n+1)}\right) = 0$, es un mínimo local.

Por lo tanto, un punto es considerado *mínimo local,* sí y solo sí su gradiente es cero y el determinante de su matriz hessiana es positivo (Weisstein, 2015). En la Tabla 4 se muestran las condiciones necesarias para diferenciar a los puntos estacionarios.

Tabla 4.

Puntos estacionarios según la matriz hessiana

Pto.	**Mínimo Local** $\boldsymbol{w_{(n+1)}}$	**Máximo Local** $\boldsymbol{w_{(n+1)}}$	**Pto. Ensilladura** $\boldsymbol{w_{(n+1)}}$
Requisito en Jacobiano	$\nabla J\left(w_{(n+1)}\right) = 0$	$\nabla J\left(w_{(n+1)}\right) = 0$	$\nabla J\left(w_{(n+1)}\right) = 0$
Requisito Matriz Hessiana	$\left\|H\left(w_{(n+1)}\right)\right\| > 0$	$\left\|H\left(w_{(n+1)}\right)\right\| < 0$	$\left\|H\left(w_{(n+1)}\right)\right\| = 0$
Forma de la función	Convexa (forma de U)	Cóncava (U invertida)	Cóncava – convexa Si $f'''\left(w_{(n+1)}\right) > 0$ Convexa – cóncava Si $f'''\left(w_{(n+1)}\right) < 0$

El entrenamiento de segundo orden es capaz de diferenciar un punto mínimo de cualquier otro estacionario, siendo ésta su principal ventaja. Su desventaja sigue siendo su incapacidad para diferenciar un mínimo local de un global, por ejemplo en la Figura 20 en el punto $w_{(n+1)}$ se ha encontrado un mínimo local que no coincide con el global.

El algoritmo de Newton y el de Levenberg-Marquardt, denominados como algoritmos de entrenamiento rápido en el Toolbox de Matlab, son considerados entrenamientos de segundo orden. Ajustan la curva del error total mediante el cálculo de su curvatura, encontrando más rápido los mínimos cuadrados no lineales. Los descensos en dirección contraria al gradiente son de mayor tamaño, agilitándose el entrenamiento. Por lo que se puede afirmar que el entrenamiento de segundo orden es mejor que el de

primer orden: es más rápido y más seguro encontrando un punto mínimo.

2.9. Control Neuronal - Neurocontrolador

Un neurocontrolador es un sistema de control desarrollado que utiliza redes neuronales artificiales, por lo que debe seguir siendo planteado como un problema de optimización no lineal. (Cotero Ochoa, 2005)

Varios son los esquemas propuestos para diseñar un neurocontrolador, haciendo una adaptación de los principios del control adaptativo al neuronal se pueden definir dos tipos fundamentales de diseño: el indirecto y el directo (Martínez Verdú, 2011).

Por un lado, en el *control neuronal indirecto* el proceso o planta no recibe una señal de control directa de la red neuronal sino del controlador. La red emula el comportamiento ideal del proceso real: imita sus características y genera señales de referencia. En función de éstas el controlador a su vez, genera la señal de control para el proceso. En el control indirecto la red neuronal y el controlador trabajan sincronizados.

Por el otro lado, en el *control neuronal directo* la red trabaja en lugar del controlador, y envía directamente señales de control al proceso mediante una configuración retroalimentada de lazo cerrado.

En el presente proyecto, el perceptrón multicapa desarrollado en Matlab reemplaza a los algoritmos de control propios de ARIA y sus salidas se envían como señales de control directas al robot.

2.9.1. Diseño de un control neuronal directo

Lo primero que se debe hacer cuando se diseña un sistema de control neuronal en esquema directo, es determinar el funcionamiento y las acciones de control que el perceptrón debe generar para gobernar sobre la

planta o proceso.

Si se dispone de un *controlador existente*, su comportamiento debe ser modelado en una red. Sus entradas y salidas coincidirían tanto en entrenamiento como en operación.

En ocasiones, al controlador lo sustituye un operador humano sin conocimiento detallado de la dinámica o del modelo del proceso. Cuando esto sucede, se recurre a un *control adaptativo directo,* diseñado libre de modelos, capaz de aprender a semejanza del operador, que encuentra criterios básicos de funcionamiento en la poca información recibida.

La principal ventaja del control adaptativo directo se da en el diseño, exige poco conocimiento previo del modelo y poco esfuerzo intelectual. Al contrario, su desventaja es su limitada aplicabilidad, siendo un diseño de adaptación y aprendizaje, necesita de un tiempo para que esto ocurra; mientras tanto la planta o proceso esta fuera de control y expuesta a fallos desastrosos. Por ejemplo, si se llegase a chocar el robot Pioneer mientras su neurocontrolador se adapta, podría averiarse.

Precisamente evitar averías y pérdidas es la misión del *control basado en un modelo*. Este esquema trabaja en un ambiente simulado, en el que se representa al proceso a través de un modelo y en el que la red pasa a ser el neurocontrolador. El robot tiene su propio simulador MobileSim que al trabajar en conjunto con Matlab, herramienta de desarrollo del neurocontrolador, facilita todo el procedimiento de modelado.

La estrategia de diseño del presente neurocontrolador respeta la directriz de los algoritmos de control propios del robot, considera los criterios que definen su comportamiento ideal en esta aplicación y debe ser modelado en un ambiente de simulación.

Una vez que se ha definido el diseño del neurocontrolador, lo siguiente que se hace es definir la configuración del sistema de control o estrategia de

control.

La Figura 21 muestra la configuración retroalimentada elegida para el sistema de control neuronal directo del robot.

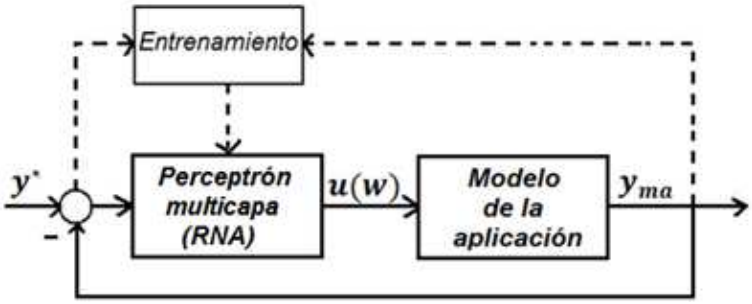


Figura 21. Control retroalimentado con red neuronal
Fuente: (Cotero Ochoa, 2005)

El entrenamiento del perceptrón multicapa es un problema de optimización no lineal, cuya formulación involucra a todos los componentes de la configuración (40). El objetivo es lograr un perceptrón con salidas óptimas que minimicen una función comparativa J respecto de los pesos w de la red.

$$RNA: \min_{\mathrm{w}} J(w) \tag{39}$$

La función comparativa J evalúa el proceso de optimización, es decir el entrenamiento. En el caso del perceptrón, el error total *E* es el encargado de esto, suma los errores de adaptación entre la señal de referencia y^* y la salida del modelo de la aplicación y_{ma}.

$$RNA: \min_{\mathrm{w}} E\{y^* - y_{ma}(u, \dots)\}\,; u = Ent(w) \tag{40}$$

La salida y_{ma} se genera en el modelo a partir de la señal de control u entregada directamente por el perceptrón. La señal u se produce en función de los pesos en la generalización de comportamientos.

La descripción matemática del sistema de control neuronal puede

también ser realizada tomando las variables propias del perceptrón. La señal de referencia del entrenamiento es el objetivo target, la salida generada es la actual, y la señal de control es la entrada.

$$RNA: \min_{\mathrm{w}} E\{targ - out_k^o(in_i, \ldots)\}\,; in_i = Ent(w) \qquad (41)$$

Cotero Ochoa (2005) concluye:

> La calidad del control obtenido con este esquema depende crucialmente de la calidad del modelo del proceso. Si un modelo no es bastante exacto, el neurocontrolador entrenado es improbable que maneje satisfactoriamente el proceso real. Si durante el entrenamiento al neurocontrolador no se le entrega suficientes datos, la generalización de comportamientos es mala.

Dichos datos son los patrones de entrenamiento que definen el modelo de referencia, si estos son insuficientes la optimización no se completa y el rendimiento del sistema es deficiente.

3. NEURAL NETWORK TOOLBOX, MEX FILES Y ARIA

En este capítulo se analizan los manuales y guías de usuario del software empleado.

Para que el lector comprenda rápidamente la aplicación, a continuación se estudia ordenadamente las herramientas y sentencias usadas. Cada una ha sido referenciada desde la documentación directamente publicada por el desarrollador, de acuerdo a su versión.

La primera sección del capítulo está dedicada al *Neural Network Toolbox* de Matlab, herramienta útil para diseñar y desarrollar el neurocontrolador, es decir la red neuronal que en esta aplicación controla a la plataforma P3-DX.

El proyecto ha sido desarrollado con el software enlistado en la Tabla 11, es decir programado con Matlab R2014a. Por indisponibilidad del respectivo manual, se ha seleccionado como bibliografía al manual de usuario completo de Neural Network Toolbox R2015a. De libre descarga del sitio web de MathWorks, está constituido por cuatro tomos:

- la guía de usuario (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015),
- la guía de arranque (Beale, Hagan, & Demuth, 2015),
- las referencias (Beale, Hagan, & Demuth, Reference Neural Network Toolbox MATLAB R2015a, 2015), y
- las notas de actualización de versiones (MathWorks, 2015).

Después de leer las *notas de actualización* se puede afirmar que entre las versiones: R2014a y R2015a no se presenta diferencia alguna que varié los comandos elegidos para la aplicación, validándose de esta manera la referencia bibliográfica citada.

Adicionalmente, respecto del Matlab R2011a, el software alternativo cuando no se dispone de Windows 7 SP1 (mirar Tabla 12), se puede asegurar que todos los programas, que han sido desarrollados para R2014a, también son íntegramente reconocidos y ejecutados en R2011a.

En la segunda sección se estudia los *MEX Files.* Con ellos se genera el interfaz MEX que permite la transferencia de datos entre el Matlab del cliente y el Aria C++ del firmware de la plataforma servidor.

Las sentencias empleadas en la generación y sus atributos son referenciados desde el manual de Interfaces Externas, divulgado por su desarrollador MathWorks en dos tomos:

- Interfaces Externas Matlab R2015a (MathWorks, 2015), y
- Referencia del API para C/C++, Fortran y Python de Matlab R2015a (MathWorks, 2015).

Adicional al manual de publicación estática, se dispone de un menú de documentación en el sitio web de soporte de MathWorks, en donde se encuentra la misma información del manual pero presentada de forma interactiva.

En la tercera y última sección se estudia la *librería de programación ARIA* de la plataforma robótica, haciendo especial énfasis en la clase ArRobot. Esta maneja la comunicación, el envió de comandos y la recepción de datos, también es el punto de partida para referenciar otras clases.

Mobile Robots, fabricante de la Pioneer P3-DX y desarrollador de ARIA, en su sitio web tiene publicado el “Manual de Referencia para Desarrolladores ARIA”. Este es un texto completo que explica el uso de

ARIA y como fluyen los datos a través de sus clases.

Además como complemento se tiene un menú interactivo de todas las clases ofertadas para el control de la plataforma. Cada clase trae un listado de sus funciones de programación, además de ejemplos donde están correctamente declaradas.

Neural Network Toolbox

3.1. Proceso de diseño y desarrollo de la Red

El diseño y desarrollo del neurocontrolador está fundamentado en un proceso de siete pasos:

- Recolección de datos
- Creación del objeto red neuronal
- Configuración de red
- Inicialización de pesos y bias
- Entrenamiento de red
- Validación de red
- Uso de red

De todo este proceso, la *recolección de datos* es el único paso que se realiza por separado antes de empezar a trabajar con el Toolbox. A la recolección se la analiza ampliamente más adelante en el capítulo 6.

Los demás pasos traen consigo sus respectivos métodos del Toolbox, que al ser invocados van ordenando lógicamente el desarrollo. Una red no puede ser usada si no ha sido creada, y no puede tener un comportamiento deseado o hacer una tarea específica si no ha sido entrenada y validada.

3.2. Creación de Red

El diseño y desarrollo empieza con la creación de la red. Cuando se invoca a este método, el software crea un objeto net que lo usa para guardar toda la información que define a la red.

3.2.1. Arquitectura de Red

En el Toolbox se tiene a disposición redes específicas de variadas arquitecturas, útiles para aplicaciones puntuales. Elegir la arquitectura correcta de la red es el primer paso en la creación.

Neurona Simple

Todas las redes, por más sencillas o complejas que sean, están construidas por neuronas simples.

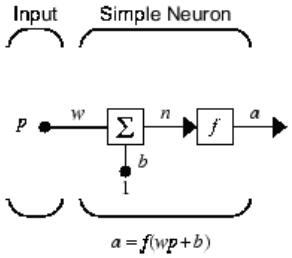


Figura 22. Neurona Simple – Toolbox
Fuente: (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

Una neurona simple tiene dos parámetros ajustables: los pesos w y el umbral b, que se usan en tres operaciones funcionales:

Función peso (weightFcn='dotprod'): es el producto wp de multiplicar la entrada p por el peso w. Cuando las entradas y las neuronas aumentan, la función peso es el producto escalar que representa la multiplicación estándar de sus matrices.

Una vez que la red ha sido creada, en la ventana de comandos de Matlab se invoca a esta función con la siguiente declaración: *>>net.layerWeights{2,1}.weightFcn*

Función entrada de red (netInputFcn='netsum'): es la suma $n = (wp + b)$ de las entradas pesadas wp y del bias b, donde el umbral puede ser considerado un peso adicional de entrada constante uno. Esta función también puede ser hallada calculando el producto entre wp y b, sin embargo la sumatoria es la operación que se usa en el perceptrón.
Invocación: *>>net.layers{1}.netInputFcn*

Función de transferencia (transferFcn): finalmente la entrada de red n es evaluada por la función de transferencia, que produce la salida escalar $a = f(wp + b)$.
Invocación: *>>net.layers{1}.transferFcn*

Capa de neuronas

Una o más neuronas simples pueden ser combinadas en una capa, y una o más capas pueden ser combinadas en una red.

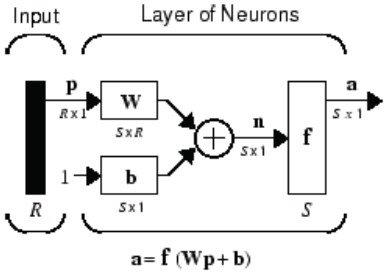


Figura 23. Capa de neuronas – Toolbox
Fuente: (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

En la Figura 23 se muestra una capa de neuronas, donde:

- R*:* es el número de elementos del vector de entrada P. Dicho de otra manera, R define la longitud del vector $P = [\,]_{Rx1}$.

- S: es el número de neuronas en la primera capa.
- $W = [\,]_{SxR}$: es la matriz de pesos que conectan los R elementos de entrada con las S neuronas de la primera capa.

$$W = \begin{bmatrix} w1,1 & w1,2 & w1,3 & \dots & w1,R \\ w2,1 & w2.2 & w2,3 & \dots & w2,R \\ w3,1 & w3,2 & w3,3 & \dots & w3,R \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ wS,1 & wS,2 & wS,3 & \dots & wS,R \end{bmatrix}_{SxR}$$

  Por ejemplo, el peso $w3,2$ es aquel que conecta a la segunda entrada con la tercera neurona de la primera capa de la red.
- $b = [\,]_{Sx1}$: es el vector bias de las neuronas.
- $a = [\,]_{Sx1}$: es el vector de salidas. Mientras más neuronas S hay en la capa, más elementos de salida conforman el vector a.

Un patrón de entrenamiento está compuesto por un juego de R elementos de entrada. Si se trata de un entrenamiento supervisado, el patrón es el vector P de entradas más los targets respectivos.

Red Multicapa alimentada hacia delante

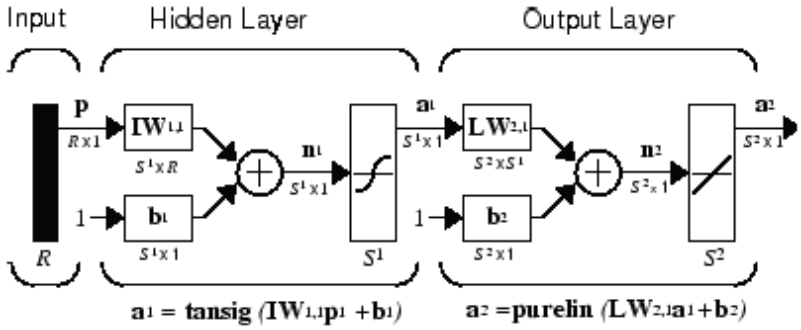


Figura 24. Red multicapa alimentada hacia delante –Toolbox
Fuente: (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

Beale, Hagan y Demuth, en la guía de usuario del Toolbox R2015a califican a "las redes neuronales multicapa alimentadas hacia delante de caballo de trabajo del Neural Network Toolbox", porque se las ocupa tanto en problemas de aproximación de funciones como de reconocimiento de patrones. Y pueden aproximar correctamente cualquier función con un

número finito de discontinuidades, siempre que tengan suficientes neuronas en la capa oculta.

Una red alimentada hacia adelante puede tener una o más capas ocultas de neuronas sigmoides, seguidas por una capa de salida de neuronas típicamente lineales, de acuerdo al respectivo criterio de aproximación de la sección 2.7.1.

3.2.2. Funciones de Transferencia (Activación)

La red *tansig/purelin* (Figura 24) es la más adecuada para resolver problemas de aproximación de funciones. Las neuronas de la capa oculta tienen una función de transferencia no lineal para poder aprender la relación no lineal entre los vectores de entrada y salida. Mientras las neuronas de la capa de salida pueden ser simplemente lineales.

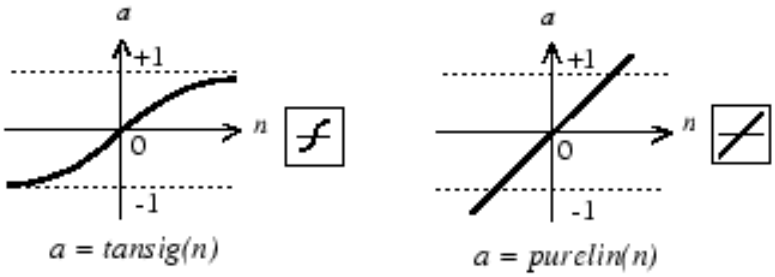


Figura 25. Funciones de transferencia - Toolbox
Fuente: (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

Función de transferencia tangente sigmoide

Entrega salidas entre menos y más uno cuando la *función entrada de red* n tiene valores entre menos y más infinito. Es común emplearla en las neuronas de las capas ocultas de la red por ser diferenciable.

Una alternativa a la tangente sigmoide es la función logarítmica sigmoide que, a diferencia de la anterior, entrega salidas solo entre cero y uno.

Función de transferencia lineal

La función '*purelin'* se la ocupa en las neuronas de la capa final, para que la red pueda ser usada como aproximador, porque así puede aprender la relación no lineal entre los vectores de entrada y salida.

3.2.3. 'feedforwardnet' - 'fitnet' - 'newff'

Después de haber definido que la arquitectura del neurocontrolador coincide con la red tansig/purelin alimentada hacia delante, se debe revisar las funciones que permiten crearla.

'feedforwardnet'

Invocación: *>>net=feedforwardnet(hiddenSizes,trainFcn)*

Capaz de soportar cualquier tipo de asignación entrada-salida, esta función es útil para aproximar funciones y también para reconocer patrones.

Toma dos parámetros: *hiddenSizes* y *trainFcn,* para crear una red alimentada hacia delante *net. hiddenSizes* es el vector fila que indica el número de neuronas en cada capa oculta por defecto, la red tiene una sola capa oculta de diez neuronas. *trainFcn* es la función de entrenamiento de la red por defecto *'trainlm'*. *net* es el objeto retornado tras la creación.

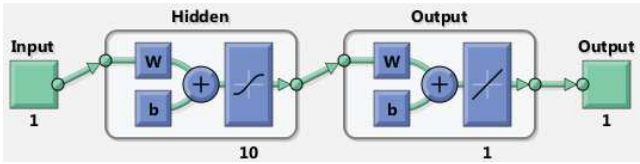


Figura 26. Feedforwardnet - Toolbox

Fuente: (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

'fitnet'

Invocación: *>>net=fitnet(hiddenSizes,trainFcn)*

Es una red *feedforwardnet* con función de transferencia tangente

sigmoide en las capas ocultas, y función lineal en la de salida. Por lo que está especialmente diseñada para resolver problemas de aproximación de funciones. Según el manual de referencias, su invocación se realiza igual a *feedforwardnet.*

'newff'

Antecesora de *feedforwardnet.* En el R2010b las funciones de creación fueron actualizadas. Si bien las funciones pasadas continúan trabajando como antes, paulatinamente son consideradas obsoletas, por lo que Matlab recomienda trabajar con las funciones actualizadas.

VISUALIZACIÓN DE RED

Invocación: >>*view(net)*

Retorna una gráfica actual de la red (Figura 26), donde se puede observar detalles como: el número de elementos de entrada y de salida, el número de capas, el número de neuronas en cada capa y las funciones de transferencia. Este método puede ser invocado en cualquier paso del desarrollo, después de haberse creado la red.

3.2.4. Objeto red neuronal

Invocación: >>*net*

El objeto (Figura 27) es usado para guardar todas las propiedades que definen a la red, en dos grandes grupos: propiedades de objeto y de subobjeto.

Propiedades de Objeto

Definen las características básicas de la red, y están organizadas en las siguientes secciones:

- General: determina las propiedades que permiten acelerar el

entrenamiento de la red.

- Arquitectura (dimensiones y conexiones): determina el número de subobjetos de la red y cómo están conectados.
- Funciones: define los algoritmos que se usan, cuando la red debe ser: adaptada, inicializada, medida en su rendimiento y entrenada.
- Valores de pesos y bias: define los parámetros ajustables de la red: matrices de pesos y vectores de bias.
- Métodos: agrupa todos los métodos que se pueden aplicar a la red cuando ya ha sido creada.

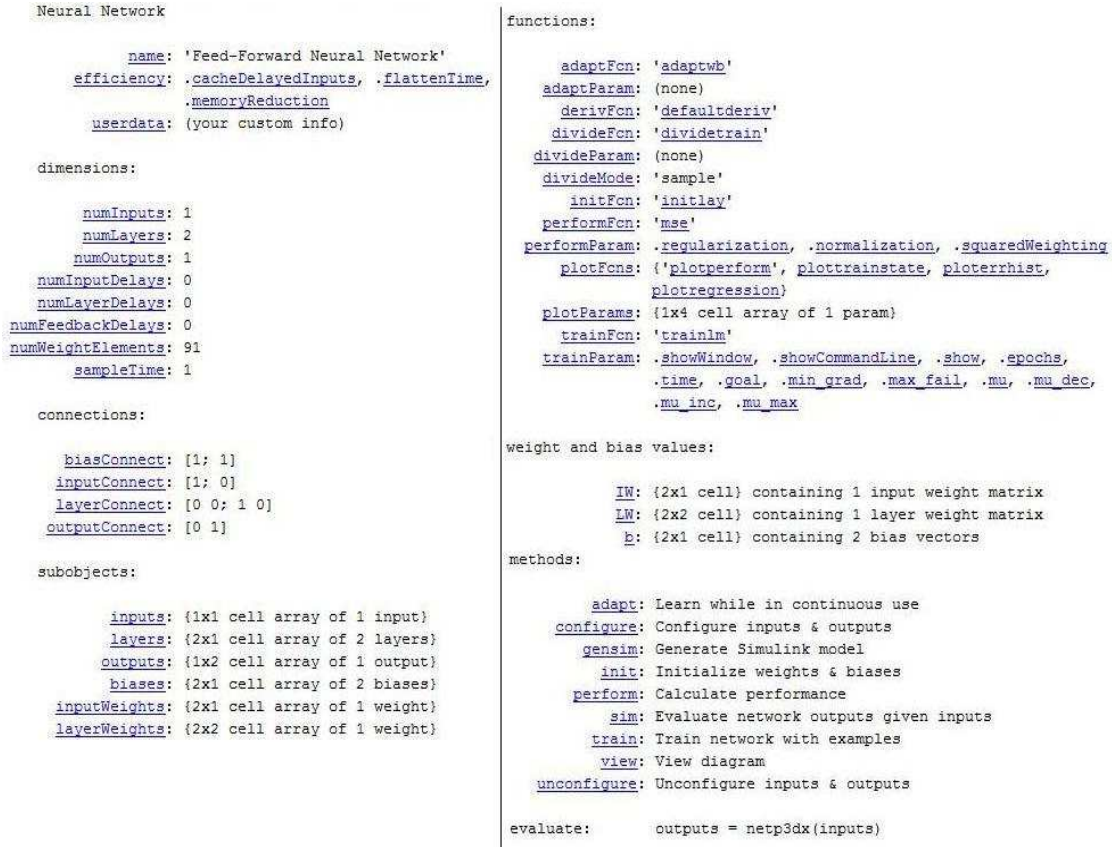


Figura 27. Objeto net creado - Toolbox

Invocación de visualización: *>>net.trainFcn*

Invocación de modificación: *>>net.trainFcn='trainoss'*

Todas las propiedades pueden ser accedidas para ser visualizadas y/o modificadas. Por ejemplo: para tener de retorno el algoritmo de entrenamiento actual, después de *net* se debe acceder con un punto a la función *trainFcn*; y para modificarla se la debe forzar al algoritmo deseado,

en este caso *'trainoss'.*

Propiedades de Subobjeto

Estas propiedades definen los detalles de los subobjetos de la red: entradas, capas, salidas, bias y pesos.

Invocación de visualización: *>>net.layers{1}.transferFcn*
Invocación de modificación: *>>net. layers{2}.transferFcn ='purelin'*

Para poder acceder a las propiedades de un determinado subobjeto, se lo debe detallar en la invocación. Por ejemplo: no es lo mismo acceder a la función de transferencia de la capa oculta que de la de salida. Para saber cuál es la función de la capa oculta, *layers{1}* direcciona a las propiedades de esta, y *layers{2} a las de la capa de salida.*

3.3. Entrenamiento de Red

Después de crear la red, se la debe configurar y luego entrenar. En la configuración se la prepara para que sea compatible con el problema a resolver, y en el entrenamiento se la somete a un proceso de sintonización de sus parámetros ajustables, para optimizar su rendimiento. Tanto en la configuración y como en el entrenamiento la red debe estar provista de datos de ejemplo. (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

3.3.1. Configuración de red - Inicialización de pesos

"Una red no configurada es automáticamente configurada e inicializada la primera vez que el entrenamiento es realizado". Alternativamente la red puede ser configurada y/o inicializada de forma manual empleando sus respectivos métodos, antes de ser entrenada. (Beale, Hagan, & Demuth, Reference Neural Network Toolbox MATLAB R2015a, 2015)

Configuración

Invocación: *>>net=configure(net,inputs,targets)*

Es el proceso que inicializa los pesos, y ajusta las entradas y las salidas, estableciendo su tamaño, rango y técnicas de procesamiento, para obtener una mejor coincidencia entre las entradas y los targets.

Inicialización

Invocación: *>>net=init(net)*

Si la red no es suficientemente precisa, se puede inicializar y reinicializar los pesos y bias antes del entrenamiento. Cada vez que se inicializa a la red, sus parámetros son diferentes produciendo soluciones diferentes. La mejor solución es aquella que ha obtenido el error cuadrático medio más bajo.

Los valores de los pesos y bias son actualizados de acuerdo a la función de inicialización de la red (*>>net.initFcn='initlay'*)

3.3.2. Método 'train'

Es el método que entrena a la red neuronal de acuerdo al algoritmo de entrenamiento *trainFcn* (sección 3.3.3) y a los parámetros de entrenamiento *trainParam* (sección 3.3.4).

Cuando es invocado este método, automáticamente se despliega una ventana (Figura 28) en la que se puede seguir todo el proceso de entrenamiento.

Invocación: *>>[net,tr]=train(net,inputs,targets)*

Tres son los atributos de entrada obligatorios: la red *net* a entrenar, la matriz *inputs* de entradas ejemplos y la matriz de *targets, de las respectivas respuestas.*

El método retorna: la red *[net]* ya entrenada, y el reporte *[tr]* en el que se registra el rendimiento del entrenamiento y las épocas generadas.

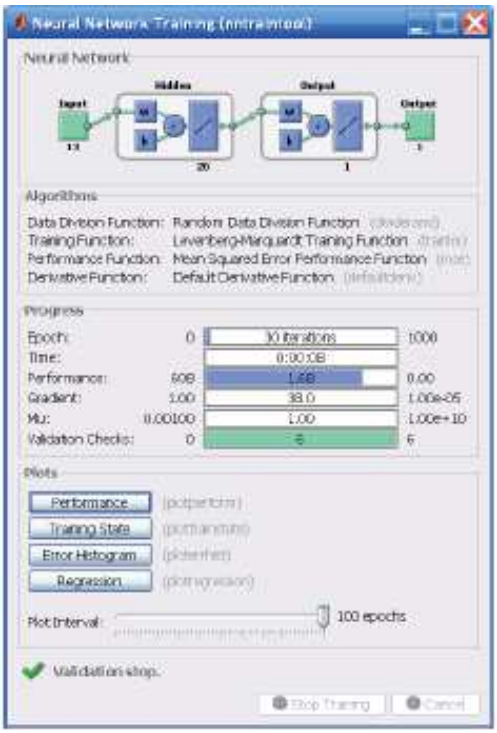


Figura 28. Entrenamiento - Toolbox

Patrones de entrenamiento

El proceso de entrenamiento requiere de un conjunto de ejemplos que reflejen el comportamiento adecuado de la red. Estos ejemplos son los patrones de entrenamiento compuestos de entradas y targets, y organizados en dos matrices:

- una de entradas $inputs = [\,]_{RxQ}$, compuesta por Q patrones de entrenamiento de R elementos de entrada, y
- una de objetivos $targets = [\,]_{LxQ}$, compuesta por Q patrones de L objetivos.

Los arreglos de entradas y targets deben ser declarados antes de configurar y entrenar la red, inclusive antes de crearla.

Proceso de Entrenamiento (performFcn='mse')

Invocación: *>>net.performFcn*

La función de rendimiento *performFcn* define como se van sintonizando los valores de los pesos y bias en la red durante el proceso de entrenamiento. La función por defecto es el error cuadrático medio *'mse'* (estudiado en la sección 2.5.2 ecuación (15)).

3.3.3. Algoritmos de entrenamiento

Invocación: *>>net.trainFcn*

Los algoritmos de entrenamiento rápido son técnicas estándar de Optimización Numérica No Lineal, fundamentados en el cálculo del gradiente de primer orden (sección 2.8.1), de segundo orden (sección 2.8.2) o de gradiente conjugada. (Babuska, 2001)

Para el entrenamiento del neurocontrolador se consideran tres algoritmos: el de la optimización de Levenberg-Marquardt *'trainlm'* y dos Quasi-Newton *'trainbfg'* y *'trainoss'.*

Método de Newton vs Métodos Quasi-Newton

El método de Newton (sección 2.8.2 ecuación (38)) es considerado un algoritmo de entrenamiento rápido. Sin embargo puede llegar a ser lento en una red neuronal alimentada hacia delante, ya que el cálculo de las segundas derivadas de la matriz Hessiana se torna complejo.

Para evitar este cálculo y acelerar el entrenamiento se han desarrollado nuevos algoritmos, conocidos como métodos Quasi-Newton o de secante, que calculan una aproximación de la matriz Hessiana en cada época, como si fuese una función del gradiente. (Beale, Hagan, & Demuth, Reference Neural Network Toolbox MATLAB R2015a, 2015).

Los métodos Quasi-Newton usan una rutina de búsqueda lineal (*>>net.trainParam.searchFcn='srchbac'*) para encontrar el punto mínimo de la función *error total* (performance) en una dirección dada (sección 2.5.2). Esta dirección siempre es negativa al gradiente del *performance* en la primera iteración, a partir de la segunda se emplean algoritmos de búsqueda propios de cada método.

Cualquier desplazamiento de búsqueda en la función es producto de haber variado los parámetros ajustables de la red, es decir los pesos. Por lo que se puede concluir, que el algoritmo de búsqueda de cada método en realidad es un algoritmo de actualización de pesos.

Algoritmo Quasi-Newton BFGS

Invocación:*>>net.trainFcn='trainbfg'*

'trainbfg' es un algoritmo de retropropagación capaz de entrenar cualquier red alimentada hacia delante actualizando los valores de los pesos y bias, conforme a la versión del método Quasi-Newton publicada por Broyden, Fletcher, Goldfarb y Shanno.

Las actualizaciones son realizadas mediante el cálculo del paso básico del método de Newton (ecuación (38)), reemplazando a la matriz Hessiana por una aproximación calculada con el jacobiano $J(w_{(n)})$:

$$w_{(n+1)} = w_{(n)} + \Delta w_{(n)} \;\; ; \;\; donde$$
$$\Delta w_{(n)} = -\frac{J^T(w_{(n)}) \cdot e(w_{(n)})}{J^T(w_{(n)}) \cdot J(w_{(n)})} \qquad (42)$$

donde:

- $J^T(w_{(n)}) \cdot e(w_{(n)}) = \mathrm{g}(w_{(n)})$ es el gradiente de la función error cuadrático medio calculado multiplicando al jacobiano transpuesto $J^T(w_{(n)})$ por el vector $e(w_{(n)})$ de los errores de la red, y
- $J^T(w_{(n)}) \cdot J(w_{(n)})$ es la aproximación de la matriz Hessiana.

Este algoritmo requiere más cálculos en cada iteración y más espacio de memoria que otros, aunque generalmente converge en un número menor de iteraciones. Es considerado un entrenamiento eficiente para redes de tamaño medio y reducido. (Beale, Hagan, & Demuth, Reference Neural Network Toolbox MATLAB R2015a, 2015).

Algoritmo de secante de un solo paso

Invocación:*>>net.trainFcn='trainoss'*

'trainoss' es otro algoritmo de retropropagación, que requiere menos cálculos por época y menos espacio de memoria que el resto de métodos Quasi-Newton, incluyendo el *'trainbfg'.*

No guarda la matriz Hessiana completa, asume en cada iteración que la Hessiana anterior ha sido la matriz identidad. Calcula las variaciones de los pesos a partir del gradiente nuevo, y de los pesos y gradientes previos. De acuerdo a la ecuación (43):

$$\Delta w_{(n)} = -\mathrm{g}(w_{(n)}) + A \cdot w_{(n-1)} step + B \cdot \Delta \mathrm{g}(w_{(n-1)}) \qquad (43)$$

donde:

- $\mathrm{g}(w_{(n)})$ es el gradiente,
- $w_{(n-1)} step$ es la variación en los pesos en la iteración anterior, y
- $\Delta \mathrm{g}(w_{(n-1)})$ es el cambio en el gradiente desde la iteración anterior.

Algoritmo Levenberg-Marquardt

Invocación:*>>net.trainFcn='trainlm'*

" 'trainlm' es a menudo el algoritmo de retropropagación más rápido en el Toolbox, y es altamente recomendado como primera opción para entrenamientos supervisados a pesar de requerir más espacio de memoria que otros algoritmos".

El algoritmo de Levenberg-Marquardt, al igual que los Quasi-Newton, ha sido diseñado para lograr la velocidad de los entrenamientos de segundo orden sin tener que lidiar con la matriz Hessiana.

La ecuación (43) calcula la variación de los pesos de acuerdo a Levenberg-Marquardt:

$$\Delta w_{(n)} = -\frac{\mathrm{g}(w_{(n)})}{J^T(w_{(n)}) \cdot J(w_{(n)}) + I \cdot mu} \qquad (44)$$

donde:

- $\mathrm{g}(w_{(n)}) = J^T(w_{(n)}) \cdot e(w_{(n)})$ es la gradiente,
- $J^T(w_{(n)}) \cdot J(w_{(n)})$ es la aproximada de la matriz Hessiana,
- I es la matriz identidad, y
- mu es un valor adaptativo escalar.

Cuando el mu es cero, este método coincide con el de Newton con matriz Hessiana aproximada (ecuación (42)). Cuando el mu es grande, se convierte en un descenso de gradiente de pasos pequeños.

Tabla 5.

Parámetros mu del 'trianlm'

>>net.trainParam	**Valor pre cargado**	**Utilidad**
.mu	0.001	valor escalar inicial del parámetro
.mu_dec	0.1	factor de decrecimiento del *mu*
.mu_inc	10	factor de crecimiento del *mu*
.mu_max	1e10	valor máximo del *mu* aceptado en el entrenamiento.

Fuente: (Beale, Hagan, & Demuth, Reference Neural Network Toolbox MATLAB R2015a, 2015)

El método de Newton es más rápido y preciso cerca de un mínimo en el error total, por lo que el objetivo es desplazarse hasta este método lo más rápido posible. Por lo tanto, el mu se reduce después de cada paso exitoso. De esta manera, el *error total* siempre se reduce en cada iteración del *'trainlm'.*

El mu es un indicador exclusivo de este algoritmo. Para su configuración se dispone de los cuatro parámetros de la Tabla 5.

Si se llega a sobrepasar el margen máximo del mu: *mu_max*, el entrenamiento es detenido.

3.3.4. Parámetros de entrenamiento

Invocación: *>>net.trainParam*

```
Function Parameters for 'trainbfg'

Show Training Window Feedback   showWindow: true
Show Command Line Feedback showCommandLine: false
Command Line Frequency                show: 25
Maximum Epochs                      epochs: 1000
Maximum Training Time                 time: Inf
Performance Goal                      goal: 0
Minimum Gradient                   min_grad: 1e-006
Maximum Validation Checks          max_fail: 6
Line search function              searchFcn: 'srchbac'
Scale Tolerance                   scale_tol: 20
Alpha                                 alpha: 0.001
Beta                                   beta: 0.1
Delta                                 delta: 0.01
Gamma                                  gama: 0.1
Lower Limit                         low_lim: 0.1
Upper Limit                          up_lim: 0.5
Maximum Step                       max_step: 100
Minimum Step                       min_step: 1e-006
B Max                                  bmax: 26
Batch Frag                        batch_frag: 0
```

Figura 29. Parámetros de 'trainbfg' - Toolbox

Para cada algoritmo hay un conjunto de parámetros modificables que direccionan el entrenamiento. Cinco de estos, que se repiten en los algoritmos, son capaces de detener el entrenamiento si alguno de sus valores es sobrepasado (Tabla 6).

Los parámetros siempre deben ser modificados antes de empezar el entrenamiento.

Invocación de visualización: *>>net.trainParam.goal*
Invocación de modificación: *>>net.trainParam.time=120*

Por ejemplo, si se entrena una red de gran tamaño, se recomienda definir un límite de tiempo. La duración predefinida es infinita, y los cálculos en redes grandes pueden tornarse complejos. Un límite de tiempo sería una opción válida para cortar un entrenamiento que está excesivamente demorado.

Tabla 6.

Parámetros de entrenamiento - Toolbox

Invocación y valor por defecto	Significado	Condición para detención
>>net.trainParam	… de entrenamiento	
.epochs= 1000	Épocas, iteraciones o repeticiones	Se ha alcanzado su número máximo
.time=Inf	Duración en segundos	Se ha superado el límite de tiempo, cuando ha sido declarado
.goal=0	(Performance) rendimiento mínimo admitido, es el error cuadrático medio (*error total*)	Cuando el error cuadrático medio es menor al *goal* declarado, el entrenamiento se detiene.
.min_grad=1e-05	(Gradient) gradiente Valor mínimo de gradiente calculado en la sintonización de pesos y bias	Para la detención, el valor del gradiente calculado en esa época debe ser menor al determinado en *min_grad.* *
.max_fail=6	(Validation Checks) Número de fallas máximas en las comprobaciones de validación	Cuando el error de validación se ha incrementado más veces de las declaradas en *max_fail*, desde la última que decreció.

*El parámetro Comprobación de Validación *Validation Checks* se activa, solo si se usa validación de red durante el entrenamiento (sección 3.4)

Fuente: (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

3.4. Validación de Red

En la validación se analiza el rendimiento de la red entrenada. Si los resultados no son satisfactorios se debe hacer modificaciones tanto en la red como en el entrenamiento, antes de un reentrenamiento.

Cuando el entrenamiento concluye ofrece indicadores analíticos y gráficos, en los cuales se resume el proceso realizado y los resultados obtenidos.

Cálculo de rendimiento de red

Invocación: *>>performance=perform(net,targets,outputs)*

'perform' calcula el rendimiento de la red, considerando la función de rendimiento (*performFcn='mse'*) elegida para el entrenamiento, por defecto el error cuadrático medio *mse*.

3.4.1. Registro de entrenamiento [tr]

Invocación: *>>tr*

Es el argumento retornado por *'train'* en segundo lugar.

```
    trainFcn: 'trainlm'
  trainParam: [1x1 struct]
  performFcn: 'mse'
performParam: [1x1 struct]
   derivFcn: 'defaultderiv'
  divideFcn: 'dividerand'
 divideMode: 'sample'
divideParam: [1x1 struct]
   trainInd: [1x354 double]
     valInd: [1x76 double]
    testInd: [1x76 double]
       stop: 'Validation stop.'
 num_epochs: 12
  trainMask: {[1x506 double]}
    valMask: {[1x506 double]}
   testMask: {[1x506 double]}
 best_epoch: 6
       goal: 0
     states: {1x8 cell}
      epoch: [0 1 2 3 4 5 6 7 8 9 10 11 12]
       time: [1x13 double]
       perf: [1x13 double]
      vperf: [1x13 double]
      tperf: [1x13 double]
         mu: [1x13 double]
   gradient: [1x13 double]
```

Figura 30. Registro de entrenamiento *[tr]* - Toolbox

En él se registran:

- todos los parámetros que han regido en el entrenamiento, como por ejemplo la función de entrenamiento 'trainFcn' y la función de rendimiento *'performFcn'*, y también
- el número de épocas producidas, entre otros datos generados.

División de patrones de entrenamiento

Invocación *>>net.divideFcn='dividerand'*

El entrenamiento, por defecto, divide aleatoriamente a los patrones en los tres conjuntos de la Tabla 7, con los porcentajes señalados.

Tabla 7.

Patrones de entrenamiento, validación y prueba

Patrones *>>net.divideParam.*	**Funcionalidad**
De entrenamiento: *.trainRatio=70/100*	Usados en el cálculo de las actualizaciones de los pesos y bias.
De validación: *.valRatio=15/100*	Usados para controlar el error de validación durante el entrenamiento. Este error normalmente decrece, al igual que el *error total* de entrenamiento, en la fase inicial. Cuando la red empieza a sobre-entrenarse, el error de validación comienza a subir durante las épocas declaradas en el parámetro *Validation Checks max_fail.* Cuando se sobrepasan estas épocas el entrenamiento es cortado.
De prueba: *.testRatio=15/100*	Usados después del entrenamiento para probar si la red generaliza correctamente. No afecta el entrenamiento

Fuente: (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

3.4.2. Gráficas de Regresión

Invocación: *>>plotregression(targets, outputs)*

El próximo paso en la validación de la red es generar una gráfica de regresión, en la cual se muestra la relación entre las salidas de la red y los targets. Si el entrenamiento fuese perfecto las salidas y los targets serian exactamente iguales, pero en la práctica la relación es rara vez perfecta.

Entrenamiento con datos de validación y prueba

Cuando se efectúa un entrenamiento con división de patrones, se dispone de un juego de cuatro gráficas de regresión (Figura 31). Para acceder a ellas se debe pulsar el botón *Regression* de la ventana de

entrenamiento.

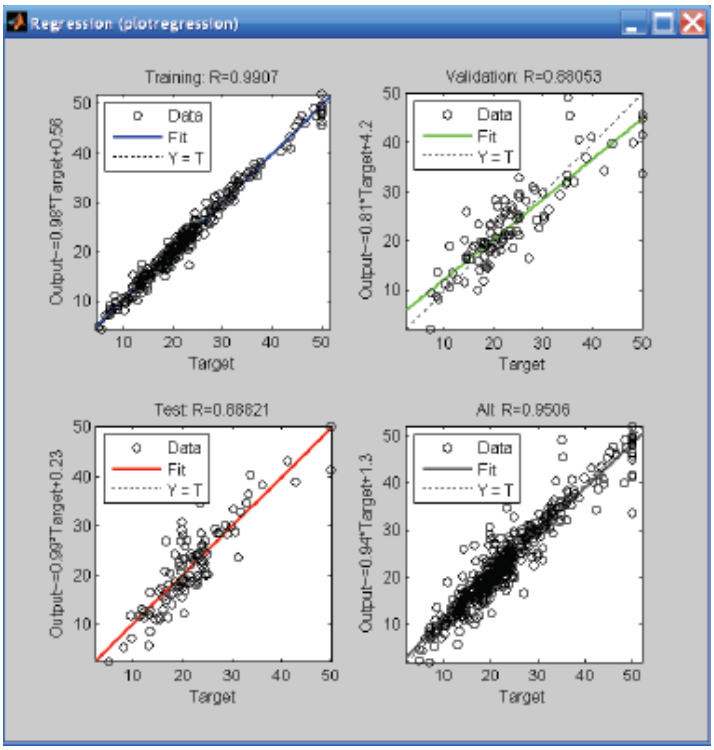


Figura 31. Gráficas de Regresión - Toolbox

De estas cuatro gráficas, tres representan al entrenamiento (azul), a la validación (verde) y a la prueba (rojo); mientras que una cuarta recoge a todos los patrones divididos.

El valor R es el coeficiente de regresión lineal e indica la relación entre los outputs y los targets. Si R=1, se indica que hay una relación lineal entre los targets y los outputs. (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

Cuando los indicadores R -global y de entrenamiento especialmente- tienen un valor cercano a uno, se puede validar el entrenamiento y la red puede ser usada.

Si por el contrario tienen un valor lejano a uno, es recomendable realizar un nuevo entrenamiento.

Entrenamiento sin datos de validación y prueba

Invocación: *>>net.divideFcn='dividetrain'*

Cuando no se dispone de una elevada cantidad de patrón es preferible emplearlos a todos en el entrenamiento, eligiendo a *'dividetrain'* como función de división. En este caso, la gráfica de regresión corresponde únicamente al entrenamiento. Una red, que ha sido entrenada sin la división de patrones, es capaz de tener el mismo rendimiento que cualquier otra entrenada con división.

Si a pesar de tener pocos patrones se decide por una división, es muy probable que el entrenamiento no se complete pues el parámetro *Validation Checks max_fail* puede ser sobrepasado con facilidad. Los cambios de función de división o la modificación de los porcentajes de los grupos, siempre deben ser declarados antes del entrenamiento de la red.

3.4.3. Nuevo entrenamiento

Tabla 8.

Modificaciones para nuevos entrenamientos

	Modificación	***Resultado Esperado***
1.	*>>net=init()*	Una inicialización y/o reinicialización de los pesos iniciales y bias, antes del entrenamiento, produce resultados distintos.
2.	*net.layers{1}.dimensions*	Una mayor cantidad de neuronas en la capa escondida, produce mejoras en la aproximación de la función.*
3.	Patrones de entrenamiento	Un incremento de datos de ejemplo produce mejoras en el entrenamiento.
4.	Inputs	Se debe incrementar el número de elementos de entrada, si aún se dispone de información relevante.
5.	*net.trainFcn*	Si las modificaciones anteriores no generan mejoras significativas, se recomienda también intentar con un algoritmo de entrenamiento distinto acorde al tamaño de la red.

* En caso de sobre-entrenamiento (rendimiento de entrenamiento bueno y específicamente rendimiento de prueba malo) se recomienda disminuir el número de neuronas.

Fuente: (Beale, Hagan, & Demuth, User's Guide Neural Network Toolbox MATLAB R2015a, 2015)

Un nuevo entrenamiento es necesario, si el rendimiento de la red entrenada no es satisfactorio o si se requieren resultados más precisos. En la Tabla 8 se exponen las modificaciones, que se recomienda realizar antes de los nuevos entrenamientos. Estas deberían ser probadas paulatinamente hasta obtener mejores resultados.

3.5. Uso de Red:

Invocación: *>>outputs=net(inputs)*

Después de que la red ha sido entrenada y validada, puede ser utilizada. El objeto *net* evalúa la salida *outputs* para una entrada dada *inputs*, generalizando la respuesta a partir de los patrones utilizados en el entrenamiento. '*inputs'* es el vector columna de entradas. La longitud de *input* debe ser la misma que la los patrones de entrenamiento, ya que ambos deben tener la misma cantidad de elementos de entrada. *outputs* es la salida, como se trata de un problema de aproximación de una función es un valor escalar.

3.5.1. Simulación de red

Invocación: *>>output=sim(net,input)*

Es un método del objeto *net* que también permite evaluar en la red una salida *output* conforme una entrada dada *input,* por lo que es una alternativa de uso de red. En este método la red es llamada como una función.

MEX Files

3.6. Definiciones Iniciales

A continuación se define cada archivo necesario para la creación de la interfaz MEX en Matlab, y para que esta pueda comunicarse con el firmware

ARCOS del robot.

A pesar de que existen en realidad dos archivos MEX, uno código fuente C++ y otro binario, la expresión “interfaz MEX” por costumbre hace referencia a ambos. Sin embargo, solo se puede programar y modificar al archivo MEX C++.

3.6.1. Archivo MEX fuente C++ (.cpp)

Es un archivo código fuente *(source code)* escrito en C++, a partir del cual se genera el archivo MEX binario. Se compone de dos rutinas: una de acceso y una de cálculo.

En este archivo se programa la interfaz misma de comunicación. En su rutina de cálculo se debe considerar a todas las funciones compuestas necesarias para el control del robot, ya sean estas de sensoramiento, de accionamiento o de configuración, el archivo es analizado en la sección 5.3.

3.6.2. Constructor MEX

Es una función, programada en Matlab, que se emplea para crear el archivo MEX binario a partir del archivo código fuente C++ existente.

3.6.3. Archivo MEX binario (.mexw32)

Es una subrutina dinámicamente enlazada que el intérprete de Matlab carga y ejecuta. Gracias a este archivo se puede llamar a cualquier subrutina C++ desde la ventana de comandos de Matlab como si esta fuese una función Matlab propia, es decir como si tuviese extensión .m (MathWorks, 2015). Por cada archivo MEX binario se puede tener solo una subrutina dinámicamente enlazada, es decir una sola función MEX.

El término *.mexw32* hace referencia a dos características del archivo:

- *mex* significa “Matlab ejecutable”, y
- *w32* indica en qué plataforma fue compilado el binario, en este caso Microsoft Windows de 32 bits. Matlab identifica a los archivos MEX binarios por la extensión de la plataforma específica. Un binario MEX compilado en una plataforma, solo puede ser ejecutado en dicha plataforma.

3.6.4. Función pasarela a archivo MEX.

Para invocar al archivo MEX, se lo debe hacer por su nombre sin la extensión *.mexw32.*

Las funciones pasarelas se emplean para invocar al archivo MEX, específicamente a las funciones de control del robot que han sido declaradas en su rutina de acceso.

3.7. Requisitos para la Creación del Archivo MEX

Para poder crear un archivo MEX binario, se debe completar los siguientes pasos en el orden indicado:

- tener instalado un compilador C++ soportado por Matlab,
- tener escrita la rutina de cálculo del archivo MEX C++,
- tener escrita la rutina de enlace -*MEX Function*- del archivo MEX C++,
- tener escrito la función *Constructor MEX*, para poder con esta compilar el archivo MEX binario a partir del archivo MEX C++, y
- compilar el archivo MEX binario y usarlo como cualquier otra función Matlab.

3.8. Selección del compilador C++

Para generar el archivo MEX binario se requiere de una versión del Matlab soportada por el compilador C++ instalado.

Los criterios de selección y compatibilidades del compilador C++ están detalladamente analizados en la sección 4.3.

3.9. Rutina de Acceso – Archivo MEX C++

Es la puerta de entrada al archivo MEX, por aquí Matlab accede a la rutina de cálculo. También es considerada la función main de dicho archivo.

Usa la función *mexFunction()* para generar el acceso. Por este motivo, muchas veces es llamada informalmente interfaz MEX. Siempre debe ser programada al final del archivo, después de toda la rutina de cálculo e incluso después de cualquier otra función.

3.9.1. mexFunction()

Sintaxis: *void mexFunction{ int nlhs, mxArray *plhs[],*
*int nrhs, const mxArray *prhs[] }*

La función MEX utiliza cuatro parámetros para enlazar los datos, dos de entrada y dos de salida (Tabla 9).

Tabla 9.

Parámetros de la función MEX

Parámetro	Descripción
prhs	Arreglo de argumentos de entrada, tipo puntero mxArray Inmodificable en el archivo MEX
plhs	Arreglo de argumentos de salida, tipo puntero mxArray
nrhs	Número de argumentos de entrada (tamaño del arreglo de entrada prhs)
nlhs	Número de argumentos de salida (tamaño del arreglo de salida plhs)

Fuente: (MathWorks, 2015)

3.9.2. Diagrama de flujo de datos

El diagrama de la Figura 32 muestra como las entradas acceden desde Matlab al archivo MEX, como las funciones *mxCreate* &y *mxGet* procesan

los datos y como se retornan las salidas a Matlab.

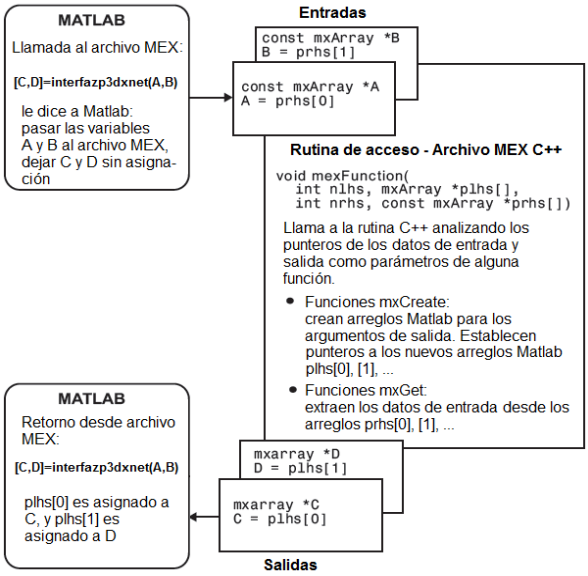


Figura 32. Flujo de datos en rutina de enlace mexFunction()

Fuente: (MathWorks, 2015)

3.10. Rutina de Cálculo

Es la otra mitad del archivo MEX C++, debe ser escrita al inicio por encima de la rutina de enlace. En ella se debe programar todos los cálculos que se quiera implementar en el archivo MEX binario.

Para la rutina de cálculo, la *mexFunction()* de la rutina de enlace es una función que se usa para validar los parámetros de entrada y convertirlos en datos requeridos (MathWorks, 2015).

3.10.1. Librerías de Referencia

Al inicio de la rutina de cálculo deben ser declaradas las librerías de soporte.

Algunas de sus funciones pueden ser usadas en la rutina de cálculo y/o de enlace del archivo MEX C++, ya sea para interactuar con los programas y variables de Matlab, o con los de Aria.

Librerías del API de Matlab

Invocación: *#include "mex.h"*

El archivo de inclusión *mex.h* da acceso a dos librerías de Matlab: Matrix y MEX. La *librería Matrix* da soporte a los arreglos mxArray, donde se guardan los datos que llegan al archivo MEX y que salen. Además permite acceder a todas las funciones que se usan en el tratamiento de estos datos, como por ejemplo las funciones: *mxGet* para extraer los datos de entrada, y *mxCreate* para crear arreglos para los datos de salida.

La *librería MEX* da soporte a todas las funciones que permiten realizar operaciones en el entorno Matlab.

Librería del API de ARIA

Invocación: *include "Aria.h"*

Este archivo de inclusión permite el acceso a todas las funciones e indicadores de las diferentes clases de la librería ARIA, que están invocadas en el archivo MEX C++; por lo que su invocación es obligatoria.

3.11. Compilación de un Archivo MEX

Invocación: *>>mex archivoMEXC++acompilar.cpp*

Invocación: *>>mex arcMEXCC.cpp incadi.h libadi.lib …*

Función mex

La función *mex* compila el archivo MEX binario. Se puede combinar múltiples archivos código fuente *(.cpp)*, archivos objeto *(.obj)* y archivos bibliotecas *(.lib)* en la compilación, simplemente listándolos con sus respectivas extensiones, separados por espacios, siempre detrás del archivo MEX C++, como se muestra en la segunda invocación. (MathWorks, 2015)

Si se compila con múltiples archivos se recomienda también el uso de la herramienta MAKE. Este crea un archivo MAKEFILE que contiene las reglas para producir archivos *.obj*, que son combinados con el binario.

ARIA

Es una librería de programación C++ que permite controlar las plataformas robóticas de MobileRobots (sección 1.7).

3.12. Clase ArRobot

Es la clase pilar de cualquier programa Aria, porque actúa:

- de punto de enlace de las comunicaciones del robot → manejando el ciclo de comunicación con el firmware del micro controlador,
- de administrador del estado del robot → receptando y accediendo a los datos de estado, y
- de sincronizador de tareas del programa añadido →decidiendo los comandos que se envían de vuelta al robot.

Además es el contenedor de todos los demás objetos de Aria, como el objeto de la clase *ArSonarDevice* denominado *sonar*.

3.12.1. Comunicación del robot

Es el primer paso a realizar en Aria. Se debe establecer una comunicación entre el objeto de la clase ArRobot y el firmware.

Todos los accesorios interna o externamente conectados usan la conexión del robot, por lo tanto sus objetos deben estar referenciados al objeto de ArRobot.

3.12.2. Conexión de Aria

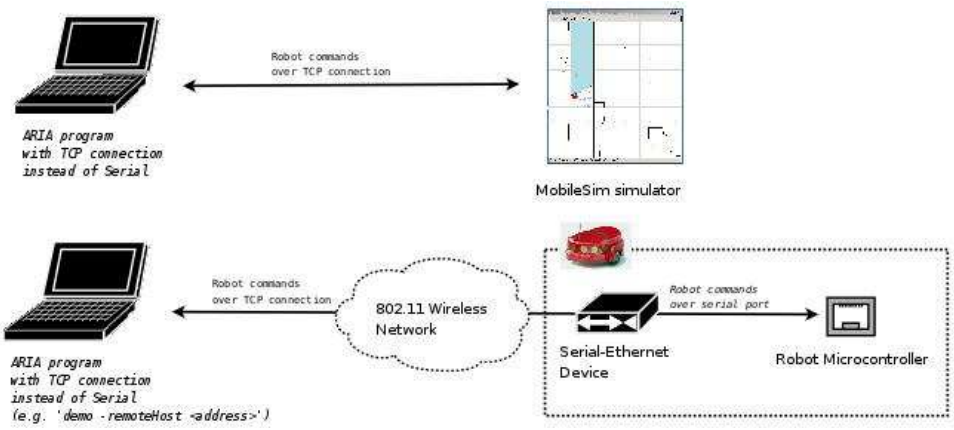


Figura 33. Conexión de Aria con el robot

Fuente: (Adept MobileRobots, 2012)

Hay varias formas de conectar un ordenador cliente (donde se está ejecutando Aria, en el caso del actual proyecto Matlab) con el servidor (el microcontrolador del robot o en su defecto al simulador MobileSim).

La Figura 33 muestra la conexión empleada en el presente proyecto. El software de control del robot desde un ordenador exterior envía comandos al microcontrolador del robot a través de una conexión TCP en el caso del simulador, o una serial en el caso del robot físico. La clase que da soporte a la comunicación es *ArSimpleConnector.*

3.12.3. Clase ArSimpleConnector (ArRobotConnector)

El objeto de esta clase es usado para establecer y manejar la conexión del robot en ambas direcciones, usando la librería Aria.

Esta clase intenta primero conectar al simulador al puerto local TCP. Si no lo consigue, entonces intenta conectar al robot al puerto serial.

Una vez conectada analiza los argumentos presentes en la línea de comandos mediante otra clase referenciada *ArArgumentParser.*

3.12.4. Clase ArArgumentParser

La clase *ArSimpleConnector* necesita información sobre los dispositivos conectados, para saber como están conectados. La clase *ArArgumentParser* obtiene toda esta información revisando los argumentos de *ArArgumentBuilder* en tiempo real.

Al iniciar la comunicación se deben utilizar argumentos precargados, llamados archivos parámetros, en la línea de comandos para que sean leídos por la ArRobot y se asegure la conexión. Caso contrario, se puede presentar un error por falta de transmisión de datos.

3.13. Datos de comunicación de ArRobot

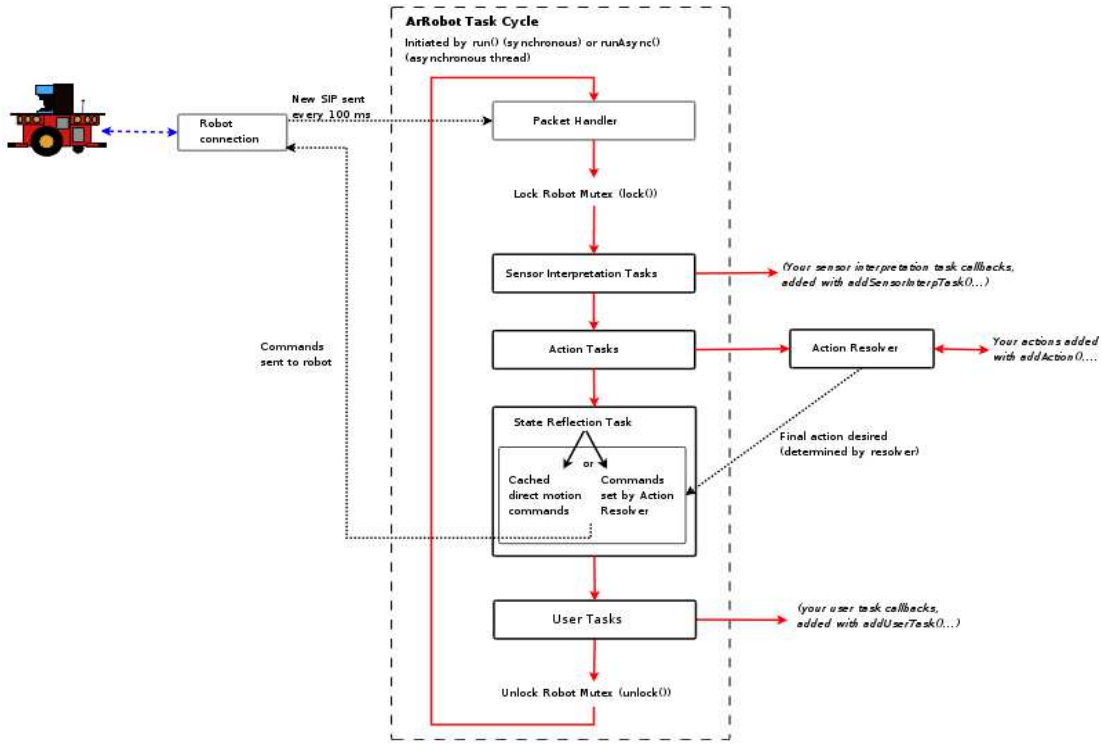


Figura 34. Flujo de datos en clase ArRobot
Fuente: (Adept MobileRobots, 2012)

La clase ArRobot envía y recibe datos en grupos separados, formando paquetes de envío y recepción.

3.13.1. Paquetes de información del servidor -SIP

Según el manual de desarrolladores ARIA, SIP son paquetes enviados cada 100 milisegundos por el servidor del robot hacia el cliente ArRobot. Contienen información actualizada sobre el robot y los accesorios, como por ejemplo las velocidades y las últimas lecturas del SONAR.

3.13.2. Paquetes de comandos

En respuesta, el programa cliente envía a través de la clase ArRobot paquetes de comandos para controlar la plataforma. En estos paquetes se puede enviar tres tipos de datos:

- *Direct Commands*: son los de más bajo nivel de acceso al robot. Son un número de comandos de un byte que están definidos de acuerdo al firmware ARCOS del microcontrolador.
- *Motion Command Functions*: de un nivel superior a los comandos directos, son comandos de movimiento simple y específicos enviados a través del flujo de datos de ArRobot.
- Actions: son objetos individuales que independientemente proporcionan solicitudes de movimiento, que a pesar de ser enviados por ArRobot ofrecen un nivel superior de control.

La idea principal de este proyecto es juntar un cliente Matlab inteligente que gobierne a la plataforma desde una computadora periférica, a través del envió de comandos sencillos al firmware. Considerando esta premisa, se limita el control de la plataforma al nivel de la clase ArRobot, sin usar *Actions*.

3.14. Clases y Funciones usadas en el proyecto

A continuación se muestra una copia de la descripción del archivo MEX C++. Todas las clases y funciones usadas en la programación del proyecto han sido directamente referenciadas en el mismo documento, para agilizar la

comprensión del código.

```
            CLASES ARIA CON SUS FUNCIONES
                empleadas en archivo
            ****************************

<CLASE>
        <Funciones>

Aria (Obligatoria)
Inicializa y desinicializa el proceso global.
     init() -> inicializa sistema operativo y datos globales
     shutdown() -> cierra todos los procesos de Aria
     parseArgs() -> analiza los argumentos para el programa
     logOptions() -> Registra todas las opciones del programa

ArRobot
Clase central para comunicación y operación del robot.
    addRangeDevice(ArRangeDevice *device) -> añade un disposiivo de
                alcance al robot y establece el puntero
    runAsync() -> inicia un nuevo hilo de procesamiento, true si no
                hay conexión con el robot el procesamiento se
                detiene.
    comInt() -> envia comando al robot con un entero como argumento
    *ArCommands -> clase que contiene los nombres de los comandos
                del microcontrolador del robot
            ENABLE -> 1 para habilitar motores, 0 para deshabilitar
            SONAR -> 1 para habilitar sonar y o para deshabilitarlo
    clearDirectMotion() -> borra los comandos directos de movimiento
    stopRunnig() -> detiene al robot de hacer más procesamiento
    checkRangeDevicesCurrentPolar() -> obtiene la lectura
                actualizada de la región polar dada, desde cualquier
                dispositivo de alcance
    getRobotRadius() -> devuelve el radio del robot en mm
    move(mm) -> mueve la distancia dada adelante o atras
    setDeltaHeading(grad) -> setea ángulo de giro, a partir de la
                posición actual
    setHeading(grad) -> setea ángulo de giro, respecto del plano
                cartesiano
    setVel(mm/seg) -> setea velocidad translacional
    setRotVel(grad/seg) -> setea velocidad rotacional
    getVel() -> retorna velocidad translacional (mm/seg)
    getRotVel() -> retorna velocidad rotacional (grad/seg)

ArRangeDevice
Clase base que retorna el alcance de todos los dispositivos de
sensoramiento.
    *ArSonarDevice -> Registra las lecturas actuales del sonar,
                como ArRangeDevice.

ArSimpleConnector
Configura la conexión de ArRobot al objeto robot.
    connectRobot() -> configura el robot y luego lo conecta

ArArgumentBuilder
Construye argumentos con datos argc y argv
    getArgv()-> obtiene argc
    getArgc()-> obtiene argv

ArArgumentParser
Configura analizador de argumentos
    loadDefaultArguments() -> añade argumentos desde archivos
                precargados
    checkHelpAndWarnUnparsed() -> advierte argumentos no analizados
```

4. INTEGRACION DE HERRAMIENTAS COMPUTACIONALES

4.1. Antecedente

Antes de empezar con el desarrollo de la aplicación, es importante poner a punto la computadora a utilizar. Si bien los requisitos en hardware no son altos, hay que poner especial atención al escoger el software a emplear, solo su correcta integración permite el funcionamiento del actual proyecto.

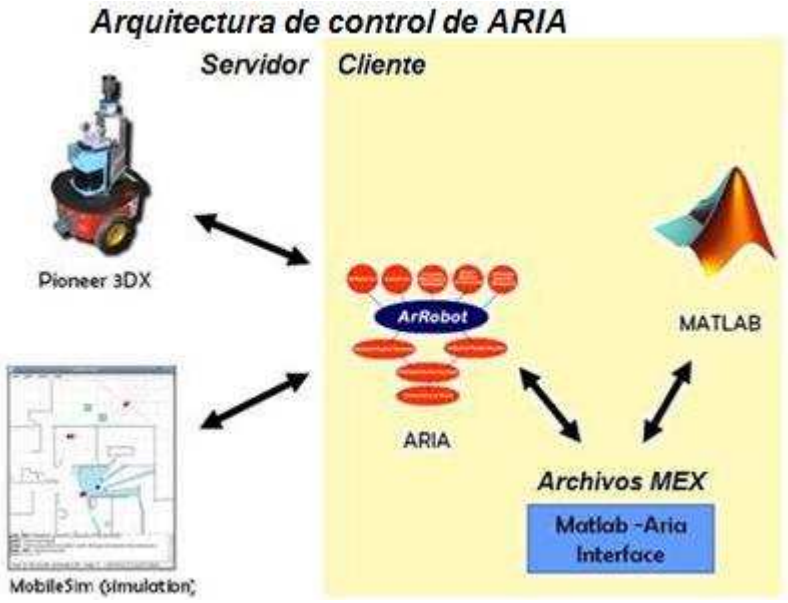


Figura 35. Interacción de las Herramientas de Software
Fuente: (Posada, 2009)

Las cuatro herramientas de software empleadas: Matlab, Microsoft Visual C++, ARIA y MobileSim, se relacionan entre sí a través del sistema operativo, en el que son ejecutadas.

La prioridad primaria en el desarrollo de este proyecto es trabajar con software propietario, específicamente con los licenciamientos disponibles en la Universidad de las Fuerzas Armadas - ESPE. Se emplea el sistema operativo Microsoft Windows y el software Microsoft Visual C++ bajo el licenciamiento de software “Microsoft Agreement” del campus de la universidad. Además se emplea la versión R2014a de Matlab y su Neural Network Toolbox, bajo la licencia adquirida por la universidad para el “Centro de Investigaciones Científicas - CEINCI”.

Se necesita de Microsoft Visual C++ para poder acceder a las librerías ARIA del robot. Además la versión elegida del Visual debe tener un compilador C++ compatible con la versión de Matlab, desde la cual se controla el robot. Por otro lado MobileSim debe ser compatible con la versión de ARIA instalada, para poder ofrecer el entorno de simulación.

En la Figura 35 se puede observar la integración del software, el cuadrado celeste “Matlab - Aria Interface” representa a la rutina de Acceso del archivo MEX, que es la compuerta de comunicación entre el lenguaje Matlab de alto nivel y el C++ de ARIA.

En el proyecto antecedente: "*Control remoto por voz del robot Pioneer P3-DX*” desarrollado por Guffanti se requirió el siguiente software:

- Sistema operativo: Windows XP, arquitectura de 32 bits
- Microsoft Visual Studio .NET 2003 (7.1), que contiene a Visual C++
- Matlab R2009a (7.8)
- ARIA 2.7.3
- MobileSim (versión sin especificar)

La Tabla 10 del Centro de Soporte de Microsoft muestra, que Visual Studio .NET 2003 puede ser instalado máximo en Windows XP, limitando su uso a este sistema operativo obsoleto.

Ningún programa o aplicación, desarrollado con esta versión de Visual,

puede ser modificado y/o recreado en sistema operativo alguno posterior a Windows XP.

Naturalmente este es un problema grave si se desean generar trabajos futuros que fortifiquen las líneas de investigación, por tal motivo la actualización de Visual Studio se antepone como una necesidad.

Tabla 10.

Requisitos para instalación de Visual Studio .NET 2003

	Visual Studio .NET 2003 (todas las ediciones)
Procesador	450 megahercios (MHz) Pentium II-procesador Procesador de 600 MHz Pentium III clase recomendado
Sistema operativo	Visual Studio .NET 2003 puede instalarse en cualquiera de los siguientes sistemas: • Microsoft Windows Server 2003 • Windows XP Professional • Windows XP Home Edition (1) • Windows 2000 Professional (Service Pack 3 o posterior requerida) (6) • Windows 2000 Server (Service Pack 3 o posterior requerida) (6) Se pueden implementar aplicaciones en los siguientes sistemas (2, 3): • Microsoft Windows Server 2003 • Windows XP Professional • Windows XP Home Edition • Windows 2000 (Service Pack 3 o posterior recomendado) • Windows Millennium Edition (Windows Me) • Windows 98 • Microsoft Windows NT 4.0 (se requiere Service Pack 6a)

Fuente: (Microsoft-Support, 2013)

Por lo tanto, en el presente proyecto se debió actualizar la versión de Microsoft Visual C++, en consecuencia fue necesario también actualizar cada una de las demás herramientas computacionales, incluyendo el sistema operativo del ordenador.

En la Tabla 11 se enlista el software, que combinado permite desarrollar el presente trabajo.

Este software fue instalado directamente en el computador de escritorio en los laboratorios de la Facultad, dicho ordenador trabaja con Windows 7 SP1.

En la Tabla 12 es enlista una segunda combinación de software, instalable en un ordenador con sistema operativo Windows Vista SP2. Esta es una alternativa a usar cuando solo se dispone de esta versión de Windows.

En ambos casos el resultado ha sido satisfactorio, se ha puesto a prueba el proyecto, consiguiendo que funcione correctamente tanto en desarrollo como en aplicación final.

Tabla 11.

Software del proyecto actual

Sistema Operativo	Windows 7 - SP2 Arquitectura de 32 bits
Microsoft Visual Studio 2010 Ultimate	(10.0), contiene a Visual C++ 2010
Matlab R2014a	(Versión 8.3)
ARIA 2.7.6	Liberado el 25 de julio del 2014
MobileSim-0.7.2-1	Liberado el 11 de junio del 2013

Tabla 12.

Alternativa para el software actual

Sistema Operativo	Windows Vista Home Basic - SP2 Arquitectura de 32 bits
Microsoft Visual Studio 2010 Ultimate	(10.0), contiene a Visual C++ 2010
Matlab R2011a	(Versión 7.12.0)
ARIA 2.7.6	Liberado el 25 de julio del 2014
MobileSim-0.7.2-1	Liberado el 11 de junio del 2013

Nótese que cada versión usada en ambas combinaciones ha sido renovada. A continuación se justifica porque se ha seleccionado cada una de ellas.

Atención: Arquitectura de 32 bits

Prestar especial atención, la arquitectura de los sistemas disponibles para este proyecto es de 32 bits. Por lo tanto, todos los programas y paquetes informáticos seleccionados, y descritos en los siguientes subcapítulos, corresponden a versiones disponibles en la misma arquitectura.

4.2. Sistemas Operativos

Después de Windows XP, Microsoft ha lanzado tres distribuciones más para ordenador: Windows Vista, Windows 7 y Windows 8, con su última versión 8.1. La ESPE dispone del licenciamiento de estas tres distribuciones.

Se ha seleccionado, como alternativa, a *Windows Vista Home Basic – Service Pack 2 (arquitectura de 32 bits)* porque de las tres distribuciones, es la de menor rendimiento y de la que más errores se han reportado. Si el proyecto funciona bien en una distribución calificada de problemática, se garantiza que también funcione correctamente en su inmediato sucesor Windows 7. Esto queda demostrado, pues es el sistema operativo elegido para el desarrollo del proyecto (mirar Tabla 11).

Que funcione en Windows 7 es una ventaja, pues le permite al proyecto estar vigente con una versión de Microsoft, que hoy por hoy es la más utilizada en el mundo de acuerdo a las últimas estadísticas globales publicadas en el sitio de la famosa herramienta de análisis de tráfico en la web “StatCount”.

Tabla 13.

Sistemas operativos para Microsoft Visual Studio 2010 Ultimate

Versión	32 bits	64 bits
Windows XP, excepto Starter Edition	Si	N/D
Windows Vista con Service Pack 2 (SP2), excepto Starter Edition	Si	Si
Windows 7	Si	Si
Windows Server 2003 SP2	Si	Si
Windows Server 2003 R2	Si	Si
Windows Server 2008 SP2	Si	Si
Windows Server 2008 R2	N/D	Si

Fuente: (Microsoft-Support-Visual-Studio, 2010)

En la Tabla 13 se demuestra, que al igual de otras tantas herramientas

computacionales, Microsoft Visual Studio tiene la versión 2010 Ultimate compatible tanto con el Windows Vista SP2 como con el Windows 7 (SP1).

Figura 36. Especificaciones del Equipo (Software alternativo)

En la Figura 36 se observan las especificaciones de software y hardware del computador empleado en la alternativa de software (Tabla 12). Se corrobora la distribución Windows Vista instalada. Además se puede ver que los requisitos en hardware son bajos: la arquitectura del sistema es de 32 bits, el procesador es un Celeron de bajo costo y rendimiento, la memoria RAM es tan solo de 1 GB y la capacidad del disco duro también es inferior.

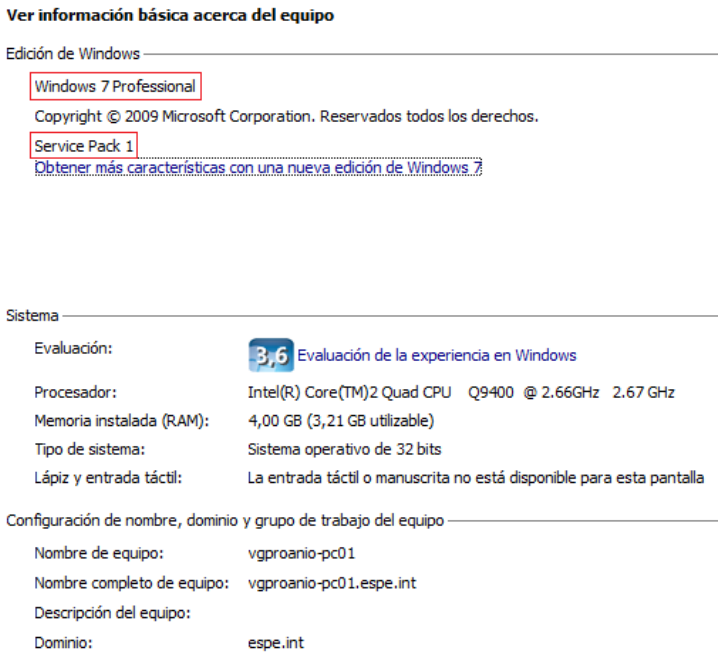


Figura 37. Especificaciones del Equipo (Software elegido)

En la Figura 37 se resume las especificaciones del equipo empleado en el desarrollo del proyecto, su sistema operativo es *Windows 7 - Service Pack 1 (arquitectura de 32 bits)*, moderno comparado con Vista SP2.

Así se demuestra, que los requerimientos para el desarrollo del proyecto son mínimos, garantizando su recreación en equipos de mejor hardware, que dispongan: de un procesador más sofisticado, por ejemplo un multinúcleo, o de una memoria RAM mayor. O utilizando software más actualizado mientras cumpla las condiciones de compatibilidad, como una versión renovada de Matlab.

4.3. Compilador C++: Microsoft Visual Studio 2010 Ultimate

Lo que motiva a utilizar una versión más actual que Microsoft Visual Studio .NET 2003 es la necesidad de trabajar con un sistema operativo vigente, que facilite la reproducción del proyecto.

Visual Studio es el principal entorno de desarrollo integrado para Windows; C++, C# y .NET son algunos de los lenguajes de programación que soporta. Por lo tanto, Microsoft Visual C++ 2010 es un componente incluido en el Visual Studio 2010.

El Visual C++ 2010 es el “ancla” en la selección del resto de programas, su instalación es obligatoria para poder acceder a las librerías de Aria 2.7.6 (ver Figura 38), la última versión publicada por MobileRobots. Además Visual Studio 2010 es compatible con las dos distribuciones: Windows 7 y Windows Vista SP2, como se mostró en la Tabla 13.

El éxito del proyecto depende de la compatibilidad entre los compiladores de C++ y Matlab, es fundamental encontrar una combinación de estos que garantice la transferencia de datos entre ellos, y que las

versiones seleccionadas compartan un mismo sistema operativo; por lo tanto se debe buscar una versión de Matlab: que disponga de un compilador compatible con Visual Studio 2010 Ultimate y que trabaje en Win 7 SP1.

Tabla 14.

Compiladores soportados por Matlab R2014a

Compiler	MATLAB *For MEX-file compilation and external usage of MATLAB Engine and MAT-file APIs*	MATLAB Compiler *For C and C++ shared libraries*	MATLAB Builder EX *For all features*	MATLAB Builder NE *For all features*	MATLAB Builder JA *For all features*	MATLAB Coder *For all features*	SimBiology *For accelerated computation*	Fixed-Point Designer *For accelerated computation*
lcc-win32 v2.4.1 *Included with MATLAB*	✓					✓ 5	✓	✓
Microsoft Windows SDK 7.1 *Available at no charge; requires .NET Framework 4.0*	✓	✓	✓	✓ 3		✓ 5	✓	✓
Microsoft Visual C++ 2013 Professional	✓	✓	✓	✓ 3		✓	✓	✓
Microsoft Visual C++ 2012 Professional	✓	✓	✓	✓ 3		✓	✓	✓
Microsoft Visual C++ 2010 Professional SP1	✓	✓	✓	✓ 3		✓	✓	✓
Microsoft Visual C++ 2008 Professional SP1 [1]	✓	✓	✓	✓ 3		✓	✓	✓
Intel C++ Composer XE 2013 [2]	✓							

Fuente: (MathWorks-Support, 2015)

En la Tabla 14 se confirma que el compilador de Matlab R2014a es compatible con Visual C++ 2010 Professional SP1. Hay que tener especial cuidado al revisar la compatibilidad, pues no solo debe existir con Matlab (la primera columna) sino también con el compilador y el constructor exterior de Matlab (segunda y tercera columnas) pues éstos permiten el acceso a las librerías compartidas para C y C++.

La última actualización de cada versión de Visual es Ultimate, por consiguiente contiene a todas las anteriores como Professional y Professional SP1. Así se demuestra, que Microsoft Visual Studio 2010 Ultimate contiene a Visual C++ 2010 Professional (también a Professional SP1) y puede reemplazarlos perfectamente.

4.4. MATLAB R2014a

Para poder avalar al Matlab R2014a se debe en primer lugar confirmar, que disponga del Neural Networks Toolbox y que también pueda ser instalado en el mismo sistema operativo que Visual Studio 2010.

De acuerdo a la guía de usuario del Toolbox publicada en 1998, esta herramienta ya estaba incluida desde esos años en Matlab.

En la Tabla 15 se puede verificar, que la versión R2014a puede ser instalada en ambos sistemas operativos: Windows Vista SP2 y Windows 7 SP1.

Tabla 15.

Sistemas operativos compatibles con Matlab R2014a

Operating Systems	Processors	Disk Space	RAM
Windows 8.1 Windows 8	Any Intel or AMD x86 processor supporting SSE2 instruction set*	1 GB for MATLAB only, 3–4 GB for a typical installation	1024 MB (At least 2048 MB recommended)
Windows 7 Service Pack 1 Windows Vista Service Pack 2			
Windows XP Service Pack 3 Windows XP x64 Edition Service Pack 2 Windows Server 2012 Windows Server 2008 R2 Service Pack 1			

Fuente: (MathWorks-Support, 2015)

Luego de estas demostraciones se puede avalar no solo el uso de Matlab R2014a sino también de Visual Studio 2010 Ultimate, para ser exacto, se debe avalar su uso conjunto como software empleado en el proyecto. Sus compiladores son compatibles y los dos pueden ser instalados en Windows 7 SP1.

Adicionalmente se debe señalar, que en el Centro de Soporte de MathWorks, (MathWorks-Support, 2015), están publicados los requisitos del sistema y los compiladores soportados para todas las versiones de Matlab.

La Tabla 16 resume sus compatibilidades con Windows: Vista SP2, 7 y 7 SP1, y con Visual C++ 2010: Professional y Professional SP1.

Tabla 16.

Compatibilidad de Matlab con Windows Vista SP2, Windows 7 y Visual C++ 2010 Professional

Matlab	*Corre en Windows Vista SP2, Windows 7 y Windows 7 SP1*	*Compatible con Visual C++ 2010 Professional y Professional SP1*
R2009b (7.9)	Sí	No
R2010a (7.10)	Sí	No
R2010b (7.11)	Sí	Sí
R2011a (7.12)	Sí	Sí
R2011b (7.13)	Sí	Sí
R2012a (7.14)	Sí	Sí
R2012b (8.0)	Sí	Sí
R2013a (8.1)	Vista SP2 & 7 SP1	SP1
R2013b (8.2)	Vista SP2 & 7 SP1	SP1
R2014a (8.3)	Vista SP2 & 7 SP1	SP1
R2014b (8.4)	Vista SP2 & 7 SP1	SP1
R2015a (8.5)	Vista SP2 & 7 SP1	SP1

Fuente: (MathWorks-Support, 2015)

A partir de R2010b todos los Matlab son una alternativa, porque traen un compilador compatible con Visual C++ 2010 Professional o Professional SP. Pero se debe considerar que las versiones posteriores al R2012b se deben instalar: o en Windows Vista SP2 o en Windows 7 SP1.

El contar con la versión Ultimate de Visual Studio es un comodín, pues contiene al SP1 de Professional, en este caso compatible con R2014a.

Adicionalmente en la Tabla 17 se enlista los compiladores soportados por Matlab R2011a.

Tabla 17.

Compiladores soportados por Matlab R2011a

Compiler	Version	MATLAB For MEX-file compilation and external usage of MATLAB Engine and MAT-file APIs	MATLAB Compiler For C and C++ shared libraries	MATLAB Builder EX For all features	MATLAB Builder NE For all features	MATLAB Builder JA For all features	MATLAB Coder For all features	SimBiology For accelerated computation	Fixed-Point Toolbox For accelerated computation
lcc - win32 *Included with MATLAB*	2.4.1	√	√				√	√	√
Microsoft Visual C++ 2010 Express *Available at no charge*	10.0	√	√	√	√ [1]		√	√	√
Microsoft Visual C++ 2010 Professional	10.0	√	√	√	√ [1]		√	√	√
Microsoft Visual C++ 2008 Professional SP1	9.0	√	√	√	√ [1]		√	√	√
Microsoft Visual C++ 2005 Professional SP1	8.0 [4]	√	√	√	√ [1]		√	√	√

Fuente: (MathWorks-Support, 2015)

4.5. ARIA 2.7.6 & MobileSim-0.7.2-1

Se debe escoger la versión de ARIA que sea compatible con el compilador C++. Para poder acceder a sus librerías, la versión de Visual C++ debe ser compatible con la de ARIA, como lo muestra la Figura 38.

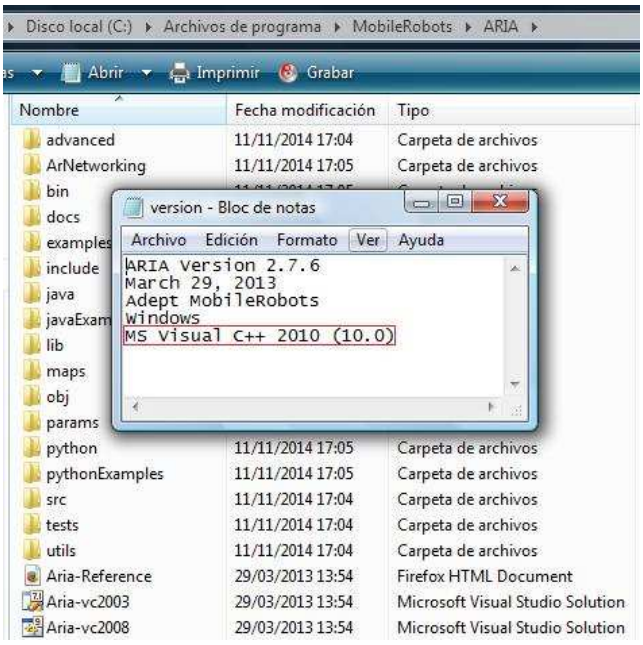


Figura 38. Carpeta de ARIA 2.7.6 instalada

Todas las dependencias ARIA, tanto para instalación, como para ejecución son librerías estándar del sistema operativo empleado y herramientas estándar de desarrollo de Visual C++.

Por otro lado se ha seleccionado MobileSim-0.7.2-1, ya que es la última versión lanzada de este ambiente de simulación. Entre sus mejoras se puede mencionar la disminución del tiempo de respuesta y también una mejora en el soporte de accesorios simulados.

4.6. Instalación de Herramientas Computacionales

De los cuatro programas que se van a instalar, dos: Visual Studio y Matlab son software propietario desarrollado y distribuido por Microsoft y MathWorks respectivamente, mientras que los otros dos: ARIA y MobileRobots son software libre.

Se debe solicitar la instalación de la licencia del Microsoft Visual Studio 2010 Ultimate, así como de los sistemas operativos Microsoft, a la “Unidad de Tecnologías de Información y Comunicación - UTIC” de nuestra universidad. En muchas ocasiones se hace tentador ocupar el 2010 Express, ya que al ser la versión liberada de Visual Studio, puede ser descargada gratuitamente desde el sitio web Centro de descargas de Microsoft, pero se debe recordar que las compatibilidades entre Matlab y Visual Ultimate, difieren con Visual Express. Se recomienda siempre trabajar con la versión Ultimate.

La instalación licenciada de Matlab R2014a, con el Neural Network Toolbox incluido, debe solicitarse al Centro de Investigaciones Científicas CEINCI.

ARIA y MobileSim son software desarrollados en código abierto por MobileRobots, su descarga es gratuita desde el mismo sitio web del fabricante. El instalador de ARIA 2.7.6 se descarga desde el centro de

descarga de ARIA, accediendo en el link: *ARIA-2.7.6.exe*. De igual manera MobileSim se descarga libremente accediendo a *MobileSim-0.7.2-1.exe*.

El orden de instalación realmente no juega un papel muy importante, sin embargo se exhorta a instalar primero Visual Studio, porque demanda más del sistema operativo. Una vez instalado, se puede seguir con Matlab y ARIA: programas que son vinculados al Visual. Finalmente se instala el entorno de simulación MobileSim.

La configuración inicial del compilador MEX en Matlab se realiza cuando Visual Studio 2010 ya está instalado (configuración descrita a profundidad en sección 5.2)

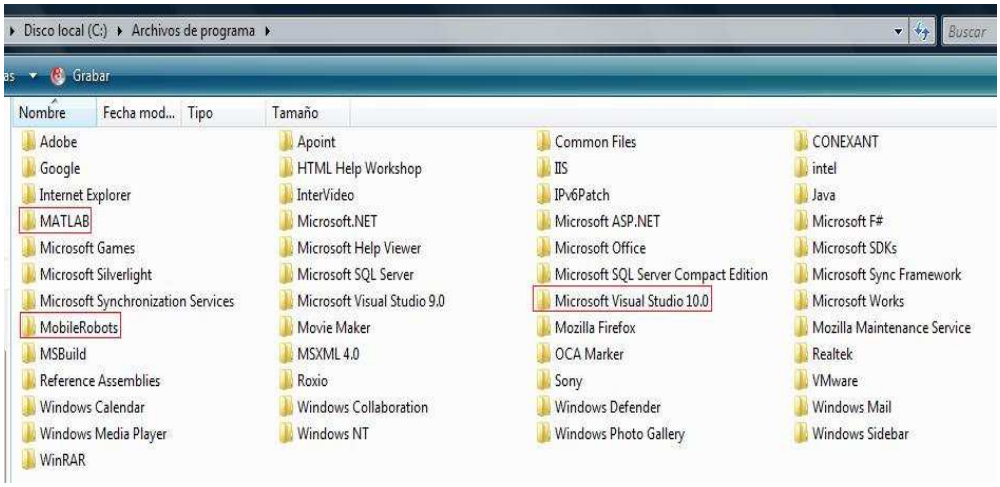


Figura 39. Archivos de Programa del equipo

Cada una de las herramientas computacionales ha sido instalada de forma "*típica"* en el path: C:\Program Files, como se puede ver en la carpeta "Archivos de Programa" en la Figura 39. Dentro de la carpeta MobileRobots se ha instalado Matlab 2.7.6 y MobileSim-0.7.2-1.

A pesar de que la instalación del software no debería presentar ninguna dificultad, si el lector necesita una guía paso a paso para instalar los programas y paquetes, se recomienda la lectura del subcapítulo "Proceso para realizar proyectos nuevos en Aria" -específicamente de la página 55 a la 64-, del proyecto antecedente "Evolución Artificial y Robótica Autónoma

desarrollada en el Robot P3-DX".

4.7. Importación de librerías faltantes de Visual

Después de haber instalado todos los cuatro programas y paquetes se debe verificar, que Visual Studio 2010 Ultimate tenga todas las librerías estáticas (.lib) necesarias para el funcionamiento del proyecto. Una librería estática es un conjunto de funciones esperando a ser llamadas por el algoritmo de control del robot, para ser enlazadas al compilador de Visual y quedar dentro de la ejecución.

El path C:\Program Files\Microsoft Visual Studio 10.0\VC\lib pertenece a la carpeta "lib" del Visual Studio 2010 Ultimate instalado. Aquí se alojan todas las librerías estáticas, en consecuencia se debe revisar si están disponibles estas tres necesarias: AdvAPI32.lib, WinMM.lib y WSock32.lib.

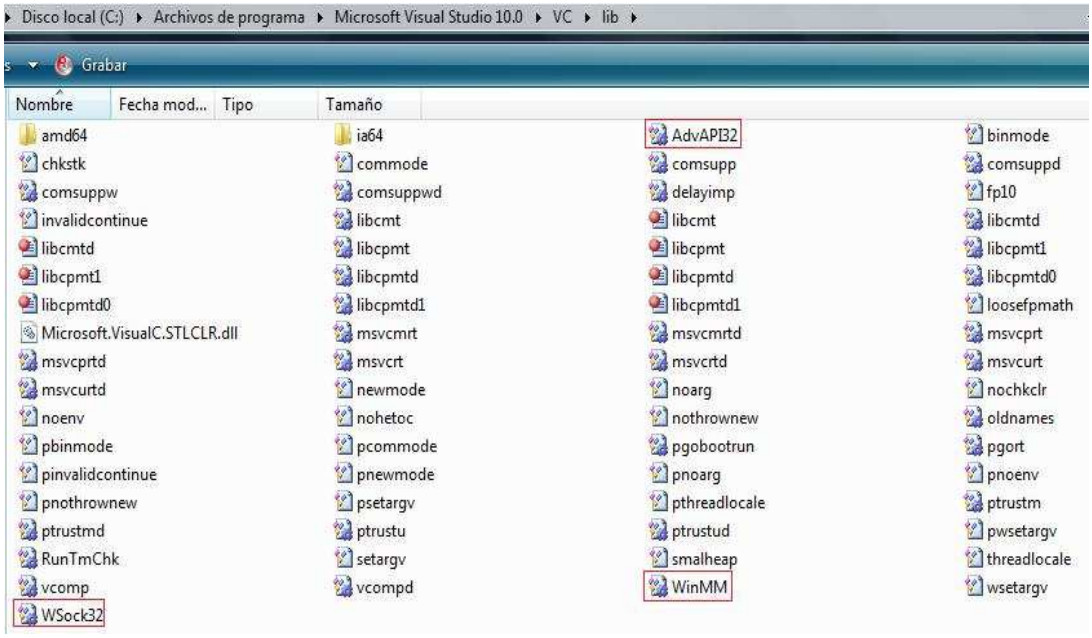


Figura 40. Librerías faltantes en el VS 2010 Ultimate

De no estar instaladas, deben ser importadas (copiadas) desde la carpeta "lib" de la plataforma SDK de Microsoft Visual Studio .NET 2003. En la carpeta CNP3DX\lib\libMV2003 están las tres librerías a disposición del lector para agilitar la reproducción del proyecto.

En la Figura 40 se muestra la carpeta “lib” del Visual Studio 2010 Ultimate instalado para la ejecución del proyecto. Como se puede ver, a parte de las librerías estáticas propias de esta versión se han copiado también las mencionadas tres. No perder de vista a estas librerías, a diferencia de las demás están escritas en mayúsculas y minúsculas: AdvAPI32.lib, WinMM.lib y WSock32.lib.

5. INTERFAZ MATLAB – ARIA

El interfaz Matlab-Aria es la puerta de enlace que permite la trasferencia de datos entre la plataforma robótica móvil Pioneer P3-DX y el neurocontrolador desarrollado en Matlab. El interfaz es capaz de entender tanto datos C++ del Aria como datos Matlab provenientes de la red neuronal.

En la Figura 41 se muestran todos los archivos necesarios para controlar al robot. El archivo MEX C++, el archivo MEX binario, el constructor MEX y las funciones pasarela, forman juntos el interfaz.

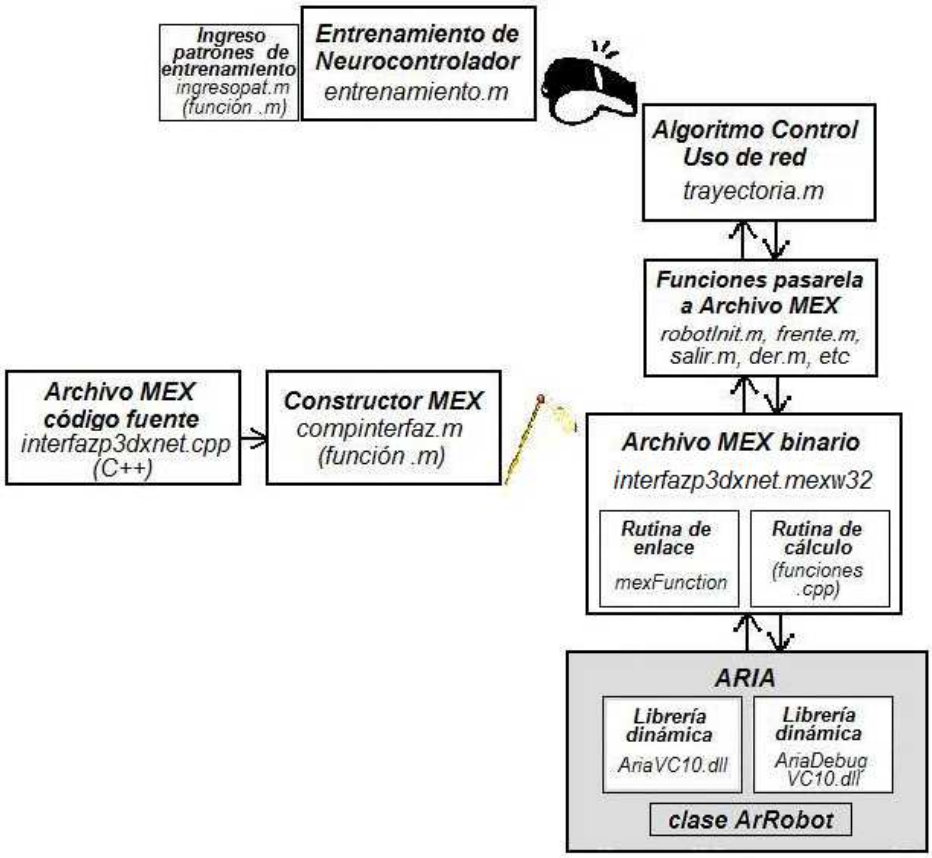


Figura 41. Diagrama global de archivos de la aplicación

5.1. Desarrollo del Interfaz

Después de haber instalado el software necesario (Tabla 11), es momento de poner a prueba su correcta integración.

El siguiente paso en el desarrollo del proyecto es la creación del interfaz, siguiendo el procedimiento establecido en el manual de Interfaces Externas del Matlab R2015a – capítulo “Intro a archivos MEX”, que se muestra en la Tabla 18.

Tabla 18.

Pasos para el desarrollo del interfaz

Pasos	Descripción	Programa
1	Tener instalado un compilador C++ soportado por Matlab	Microsoft Visual Studio C++ 2010
2	Escribir la rutina de cálculo en el archivo MEX C++	*interfazp3dxnet.cpp* (primera parte)
3	Escribir la rutina de enlace/acceso en el archivo MEX C++	*interfazp3dxnet.cpp* (segunda parte)
4	Escribir la función constructor MEX	*compinterfaz.m*
5	Compilar archivo MEX binario y usarlo como cualquier función integrada Matlab	*Interfazp3dxnet.mexw32* (Funciones pasarela: *robotInit.m*, *salir.m*, etc)

Fuente: (MathWorks, 2015)

La meta del proceso es obtener un archivo MEX binario, para usarlo en Matlab como cualquier otra función incorporada (definiciones en sección 3.6).

Atención: Aplicación desarrollada en Matlab R2014a

En el presente documento, a partir de esta sección, todos los archivos analizados fueron desarrollados en Matlab R2014a, software enlistado en la Tabla 11. Además esta aplicación ha sido probada también en el R2011a, software alternativo enlistado en la Tabla 12, obteniendo los mismos resultados positivos.

Recuerde: siempre que se ocupe otras versiones de software, se debe primero editar las rutas de acceso *paths* de todos los archivos (cabecera *headers,* librerías dinámicas *dll* y estáticas *lib*) declarados en el constructor MEX, para luego volver a crear el archivo MEX binario dinámicamente enlazado con los nuevos archivos especificados en los *paths* (en detalle sección 5.4.2).

5.2. Configuración del compilador C++

Se debe seleccionar como compilador *mex* de Matlab al compilador C++ incluido en el Microsoft Visual Studio 2010 Ultimate.

5.2.1. Identificación del Compilador

Para saber si Matlab ya lo ha configurado como compilador *mex* a usar, hay que preguntárselo directamente a través de la ventana de comandos con la invocación señalada.
Invocación: *>>mex.getCompilerConfigurations('C', 'Selected')*

5.2.2. Cambio de Compilador precargado

Invocación: *>>mex –setup*

```
Please choose your compiler for building MEX-files:

Would you like mex to locate installed compilers [y]/n? n

Select a compiler:
[1] Intel C++ 11.1 (with Microsoft Visual C++ 2008 SP1 linker)
[2] Intel Visual Fortran 11.1 (with Microsoft Visual C++ 2008 SP1 linker)
[3] Intel Visual Fortran 11.1 (with Microsoft Visual C++ 2008 Shell linker)
[4] Lcc-win32 C 2.4.1
[5] Microsoft Visual C++ 6.0
[6] Microsoft Visual C++ 2005 SP1
[7] Microsoft Visual C++ 2008 SP1
[8] Microsoft Visual C++ 2010
[9] Microsoft Visual C++ 2010 Express
[10] Open WATCOM C++

[0] None

Compiler: 8

Your machine has a Microsoft Visual C++ 2010 compiler located at
C:\Program Files\Microsoft Visual Studio 10.0. Do you want to use this compiler [y]/n

Please verify your choices:

Compiler: Microsoft Visual C++ 2010
Location: C:\Program Files\Microsoft Visual Studio 10.0
```

Figura 42. Configuración de compilador C++

Si aún no es reconocido, se lo debe configurar mediante el dialogo que se genera con esta invocación (Figura 42).

Si entre las opciones de compiladores C++ disponibles no se encuentra el Visual C++ 2010 (Ultimate), regresar a la sección 4.6 y verificar que la instalación de este software haya sido exitosa.

5.3. MEX File C++

Ubicación del programa: CNP3DX \ src\interfazp3dxnet.cpp

Tanto la rutina de cálculo como la de acceso están incluidas en el mismo archivo MEX con código fuente *interfazp3dxnet.cpp.*

Este archivo puede ser programado directamente en un *script* del editor de Matlab, o en cualquier otro entorno de programación que soporte lenguaje C++, como el Visual Studio 2010.

Mientras el archivo no presente problemas al ser compilado en Matlab por la función *Constructor MEX*, no hay ningún tipo de restricción con respecto al entorno.

5.3.1. Rutina de cálculo

Es el código fuente que le da la funcionalidad al archivo MEX binario, todo lo programado aquí puede ser usado luego en Matlab (definición ampliada en sección 3.10)

La rutina de cálculo en el proyecto está conformada por ocho funciones compuestas, útiles para el control del robot. Dichas funciones han sido escritas lógicamente en lenguaje C++, respetando el convenio de código del API de Aria.

En esta parte del documento únicamente se enlista las funciones declaradas en esta rutina. Posteriormente en la sección 5.5 se revisa el uso de objetos, clases y funciones de Aria.

Librería de Referencia y Contenedores

Al inicio del código deben ser invocadas las librerías del API de Aria y del API de Matlab, mediante sus archivos de inclusión (sección 3.10.1). Seguido, se deben declarar los contenedores de los objetos globales a usar en las funciones.

```
// *** Inclusión de librerías de referencia Aria y Matlab****
// ---------------------------------------------------------
// C:\Program Files\MobileRobots\ARIA\include\Aria.h
// C:\Program Files\MATLAB\R2014a\extern\include\mex.h
#include "Aria.h"
#include "mex.h"

// *** Objetos  Globales *** (robot, conexión y sonar)
// -------------------------
ArRobot robot;
ArSimpleConnector *simpleConnector;
ArSonarDevice sonar;
```

De este punto en adelante están escritas las funciones compuestas, ordenadas en tres grupos:

Funciones de Configuración

Son dos funciones que envían órdenes al robot, sin ser receptoras de la señal de control del neurocontrolador.

myRobotInit() enciende y conecta al robot al iniciar el algoritmo de control.

```
void myRobotInit(){
// *** Recepción y envio de datos ***
    ArArgumentBuilder argBuilder;
      char **argV = argBuilder.getArgv();
      int  argC = (int)argBuilder.getArgc();
    // *** Inicialización Global***
    Aria::init();
    // *** Comprobación de Conexión ***
      ArArgumentParser parser(&argC, argV);
    // Carga analizador a la conexión
```

```
    simpleConnector = new ArSimpleConnector(&parser);
      parser.loadDefaultArguments();
    // Carga el robot a la conexión
    if (!simpleConnector->connectRobot(&robot))
      {
            printf("Robot no conectado...\tSaliendo\n");
            Aria::shutdown();
      }
      if (!Aria::parseArgs() || !parser.checkHelpAndWarnUnparsed(1))
      {
            printf("Falla de analizador de argumentos\n");
        Aria::logOptions();
            Aria::shutdown();
      }
    //Carga el sonar al robot para poder recibir los datos
    robot.addRangeDevice(&sonar);
    // *** Creación del método run ***
      robot.runAsync(true);
    // *** Código a ejecutar ***
      robot.comInt(ArCommands::ENABLE, 1); //enciende motores
    robot.comInt(ArCommands::SONAR, 1); } //enciende sonar
```

mySalir() , en cambio, lo apaga y lo desconecta al terminar la aplicación o cuando el robot está en estado de emergencia (a punto de chocar).

```
void mySalir(){
      robot.clearDirectMotion(); // borra comandos directos
      robot.stopRunning(); //para procesamiento
// *** Desconexión del robot ***
      Aria::shutdown(); }
```

Funciones de Sensoramiento

Son las encargadas de retornar las lecturas actualizadas del sonar al neurocontrolador. Es decir, estas producen las señales de sensamiento.

```
/* Ángulos para barrido polar:
  90 50 30 10 -10 -30 -50 -90 -90 -130 -150 -170 170 150 130 90
                        front
                       10  -10
              30  3     4 5   6  -30
                    \    |#|   /
             50  2 \ \   |#|  / / 7  -50
                    \ ####### /
          90  1 ---[]#### ####[]--- 8  -90
         90  16 ---[]###   ###[]--- 9  -90
                    / ####### \
           130  15 / /   | |   \ \ 10  -130
                  /     | |     \
             150  14    13 12    11  -150
                     170   -170
                        back */

// (10) Lectura sonar izquierda (70°,90°)
// ------------------------------------------------------------
void mySonarIzq(double output[]){
    output[0]=robot.checkRangeDevicesCurrentPolar(70,90)-
robot.getRobotRadius(); }

// (11) Lectura sonar frente (-10°,10°)
```

```
// ---------------------------------------------------------------
void mySonarFrente(double output[]){
    output[0]=robot.checkRangeDevicesCurrentPolar(-10,10)-
robot.getRobotRadius(); }

// (12) Lectura sonar derecha (-70°,-90°)
// ---------------------------------------------------------------
void mySonarDer(double output[]){
    output[0]=robot.checkRangeDevicesCurrentPolar(-90,-70)-
robot.getRobotRadius(); }
```

Funciones de Actuación

Son las encargadas de generar los movimientos del robot, conforme la señal de control que reciben del neurocontrolador.

```
// (20) Movimiento 1: giro 30° izq y avanza 200mm
// ---------------------------------------------------------------
void myIzq(){
    robot.setDeltaHeading(30);
    robot.move(200); }

// (21) Movimiento 2: avanza 350 mm en línea recta
// ---------------------------------------------------------------
void myFrente(){
    robot.move(350); }

// (22) Movimiento 3: giro 30° der y avanza 200mm
// ---------------------------------------------------------------
void myDer(){
    robot.setDeltaHeading(-30);
    robot.move(200); }

// (23) Movimiento alterno 1: giro 30° izq
// ---------------------------------------------------------------
void myGIzq(){
    robot.setDeltaHeading(30); }

// (24) Movimiento alterno 2: giro 30° der
// ---------------------------------------------------------------
void myGDer(){
    robot.setDeltaHeading(-30); }
```

Nota: los movimientos alternos se usan solo para mover al robot a una posición inicial específica.

5.3.2. Rutina de acceso – mexFunction()

Es la puerta de entrada al archivo MEX, a través de ella Matlab accede a la rutina de cálculo (todas las definiciones de la rutina y de la *mexFunction()* disponibles en la sección 3.9)

La rutina de acceso del proyecto, para el análisis, es dividida en tres secciones:

1era: Creación de la rutina, con mexFunction()

Se invoca a la función *mexFunction()*, la que con sus parámetros envía y recibe datos.

```
void mexFunction( int nlhs, mxArray *plhs[],
              int nrhs, const mxArray*prhs[] )
{
      // Declaración de variables
      double *output; // de envió
      int myRobotFunction; // selector de recepción
      double *firstArg, *secondArg;// receptor
```

2da: Verificación de parámetros MEX-File de entrada

A pesar de que en todas las funciones programadas se ocupan máximo dos arreglos de entrada *prhs[0]* y *prhs[1],* se ha verificado una tercera entrada *prhs[2]* y se ha declarado una variable para esta.

```
    // Verificación de los parámetros MEx File de entrada
    myRobotFunction = mxGetScalar(prhs[0]);
    if (nrhs >= 2){
    firstArg  = mxGetPr(prhs[1]);
    }
    if (nrhs >= 3){
    secondArg = mxGetPr(prhs[2]);
    }
```

3ra: Selector de Funciones

Es una lista de casos que agrupa todas las funciones generadas para controlar el robot. Cada función es evaluada por el arreglo *prhs[0],* a través de la variable *myrobotFunction.*

```
    switch(myRobotFunction){

// **Funciones de Configuración**
    case 0:
        // Encendido de robot (sensores y motores)
      myRobotInit();
      break;
    case 1:
        // Apagado de robot
```

```
      mySalir();
      break;
// **Funciones de Sensoramiento**
    case 10:
        // Lee sensor izquierdo (70°,90°)
        plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
        output = mxGetPr (plhs[0]);
        mySonarIzq(output);
        break;
       case 11:
        // Lee sensor frontal (-10°,10°)
        plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
        output = mxGetPr (plhs[0]);
        mySonarFrente(output);
        break;
    case 12:
        // Lee sensor derecho (-70°,-90°)
        plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
        output = mxGetPr (plhs[0]);
        mySonarDer(output);
        break;
// **Funciones de Actuación**
    case 20:
        // Movimiento 1: giro izq y avance
        myIzq();
        break;
    case 21:
        // Movimiento 2: avance línea recta hacia delante
        myFrente();
        break;
    case 22:
        // Movimiento 3. giro der y avance
        myDer();
        break;
    case 23:
        // Movimiento alterno 1: giro izq
        myGIzq();
        break;
    case 24:
        // Movimiento alterno 2: giro der
        myGDer();
        break;
    case 2:
        // Movimiento alterno recto
        // Movimiento en linea recta una distancia +/- mm
        robot.move(*firstArg);
        break;
    default:
    mexErrMsgTxt("Función no implementada...");
}
```

Nota: los movimientos alternativos se usan solo para mover al robot a una posición inicial específica.

5.3.3. Flujo de datos a través del archivo MEX C++

A continuación se ejemplifica el envío y la recepción de datos mediante el análisis de dos funciones, una de solo recepción, y una de recepción y respuesta.

Flujo a través de una función receptora

Un ejemplo de función receptora es *move(),* solo recibe la orden de mover el robot una distancia en milímetros, y no retorna ningún dato.

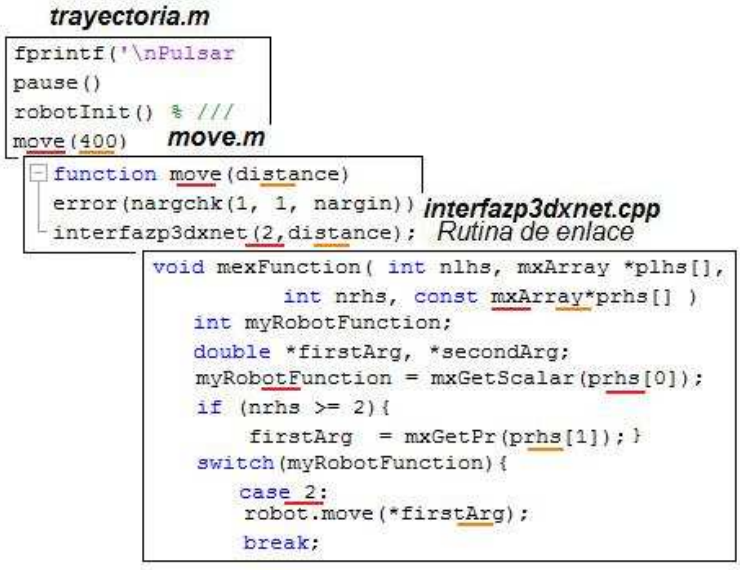


Figura 43. Flujo de datos en recepción

En modo manual se invoca la función *move.m*, con un atributo de entrada *distance* de valor escalar 400.

A su vez, esta función de enlace *move.m* invoca al archivo MEX *interfazp3dxnet.cpp,* como si fuese otra función Matlab incorporada, con dos atributos de entrada: el primero un indicador escalar 2 asignado y el segundo el escalar distance.

La rutina de enlace del archivo MEX recibe a este par de atributos con las siguientes asignaciones:

- indicador 2 → **prhs[0],* y
- escalar distance → **prhs[1]*.

**prhs[0]* y **prhs[1]* son dos vectores que contienen punteros a los arreglos de argumentos mxArray del archivo MEX. (MathWorks, 2015)

Mediante la función *mxGetScalar(const mxArray *pm)* se obtiene el componente real del primer elemento del arreglo *prhs[0];* así el indicador real 2 es recuperado y es registrado en la variable *myRobotFunction.* Mediante la función *mxGetPr(const mxArray *pm)* se obtienen los elementos reales de tipo doble del arreglo *prhs[1]*; así se recupera el valor real cargado en distance, y se lo registra en la variable *firstArg.* (MathWorks, 2015)

Al 2 de la variable *myRobotFunction* se lo evalúa en la estructura switch, y activa la función primitiva *move* de la clase ArRobot de Aria. Y a la distance 400 se la envía como atributo de esta función *move.* (Adept MobileRobots, 2012)

Flujo a través de una función de recepción y respuesta

A diferencia de la función anterior, *sonarFrente.m* a más de tener un atributo asignado de entrada tiene también un atributo de salida: el arreglo *[sensorArray]*.

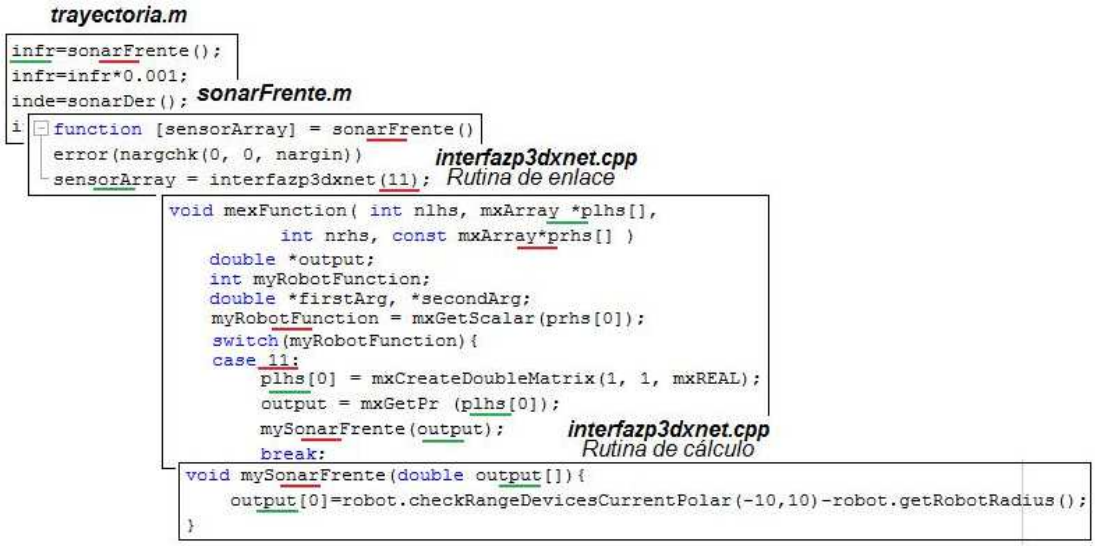


Figura 44. Flujo de datos en recepción y envio

El atributo asignado 11 activa en la rutina de acceso la función compuesta *mySonarFrente(double output[]),* que es ejecutada en la rutina de cálculo del archivo MEX. Esta función devuelve en la variable *output* la

última lectura del SONAR de una determinada región.

Mediante la función *mxCreateDoubleMatrix(número de filas, número de columnas, mxReal)* se crea el arreglo tipo doble mxArray *plhs[0]* donde se guarda la salida *output,* y se le asigna un puntero. (MathWorks, 2015)

A través del arreglo *plhs[0]* del archivo MEX, el valor real *output* es enviado al arreglo *[sensorArray]* de la función Matlab *sonarFrente.m.*

5.4. Constructor MEX

Ubicación del programa: CNP3DX \compinterfaz.m

Una vez que el archivo MEX C++ está totalmente escrito, debe ser compilado por el constructor MEX *compinterfaz.m,* para crear el archivo MEX binario *interfazp3dxnet.mexw32*.

El constructor está compuesto por dos fracciones de código, de vital importancia para el proyecto: la actualización de rutas de acceso *paths* y la compilación combinada. De la correcta declaración de estas, depende el éxito de los enlaces dinámicos del archivo binario.

5.4.1. Archivo MEX binario

Ubicación del programa: CNP3DX \bin\interfazp3dxnet.mexw32

Es una subrutina dinámicamente enlazada al intérprete de Matlab, que permite cargar y ejecutar todas las funciones declaradas en el archivo MEX C++ *interfazp3dxnet.cpp* (sección 3.6.3)

El MEX binario es una subrutina dependiente, es decir solo puede ser usada en la plataforma en que ha sido compilada. Además esta enlazada a los archivos declarados en su compilación.

Un archivo MEX binario solo puede ser usado en un computador

diferente, cuando este tiene el mismo sistema operativo y la misma arquitectura, y cuando los archivos enlazados por compilación tienen las mismas rutas de acceso *paths,* que las declaradas en el constructor MEX.

Es poco probable que se den tantas coincidencias, especialmente con los *paths*. Por lo tanto, siempre que se desee recrear este interfaz, en primer lugar se debe actualizar los *paths* declarados en el constructor MEX y luego se debe sustituir al viejo archivo MEX binario *interfazp3dxnt.mexw32* por uno recién compilado.

5.4.2. Actualización de rutas de acceso path

Todos los archivos, a los que pertenecen las rutas *paths* aquí declaradas, son integrados en la compilación al archivo MEX binario para que este pueda enlazarlos dinámicamente.

```
function compinterfaz
% Actualización de Paths
% ======================
% Archivos de Aria Version 2.7.6 (MS Visual C++ 2010)
% ---------------------------------------------------
% Include Files - Headers
inc{1} = 'C:\Program Files\MobileRobots\ARIA\include';
% Library Files
lib{1} = 'C:\Program Files\MobileRobots\ARIA\lib\AriaVC10.lib';
lib{2} = 'C:\Program Files\MobileRobots\ARIA\lib\AriaDebugVC10.lib';
% Compiler Files
% Si winmm.lib, Wsock32.lib,AdvAPI32.Lib copiados car. del Visual
C++ 2010 (sección 4.7.)
% lib{3} = 'C:\Program Files\Microsoft Visual Studio
10.0\VC\lib\WinMM.Lib';
% lib{4} = 'C:\Program Files\Microsoft Visual Studio
10.0\VC\lib\WSock32.Lib';
% lib{5} = 'C:\Program Files\Microsoft Visual Studio
10.0\VC\lib\AdvAPI32.Lib';
% Para winmm.lib,Wsock32.lib,AdvAPI32.Lib integrados car. CNP3DX
lib{3} = 'C:\CNP3DX\lib\libMV2003\WinMM.Lib';
lib{4} = 'C:\CNP3DX\lib\libMV2003\WSock32.Lib';
lib{5} = 'C:\CNP3DX\lib\libMV2003\AdvAPI32.Lib';
% El binario se genera correctamente con ambos paths.
```

Archivos Visual Studio enlazados

Son tres librerías estáticas de compilación. En la sección 4.7, se indicó que estas son importadas de una versión anterior del Microsoft Visual Studio.

Se recomienda copiar estas librerías o en la carpeta de librerías estáticas del Visual 2010 instalado, o por motivos prácticos en la carpeta *C:\CNP3DX\lib\libMV2003* del proyecto. Nuevamente, no olvidar actualizar las rutas de acceso de acuerdo a su elección.

Archivos ARIA enlazados

Tabla 19.

Archivos ARIA enlazados al archivo MEX binario

Tipo	Archivos	Descripción
De inclusión	*Aria.h, ArRobot.h, ArRangeDevice,…* (todos los archivos en carpeta include ARIA)	Integrados para que el binario pueda acceder a través de los identificadores a cualquier función de cualquier clase de la librería ARIA.
Librerías	*AriaVC10.lib* *AriaDebugVC10.lib*	Integrados para señalizar la conexión entre Aria y el compilador C++ del Microsoft Visual C++ 2010

5.4.3. Compilación combinada

```
% Construcción de argumentos
% Include files
incFiles = [];
for k = 1:size(incShort,2)
    incFiles = [incFiles ' -I',incShort{k}];
    fprintf('Construcción de Argumentos de Include File: %d\n',k)
end
% library files
libFiles = [];
for k = 1:size(libShort,2)
    libFiles = [libFiles, ' ', libShort{k}];
    fprintf('Construcción de Argumentos de Library File: %d\n',k)
end


% Compilación
cmd{1} = ['mex src\interfazp3dxnet.cpp -outdir bin -O ',incFiles,
libFiles];
for k = 1:size(cmd,2)
   fprintf('Compilando archivo MEX binario %d\n',k)
   eval(cmd{k})
end
```

La función *mex* compila al archivo *interfazp3dxnet.cpp* de la carpeta *src,* enlazándolo con los archivos *include* y *lib* declarados (sección 3.11). Además, coloca al archivo generado *interfazp3dxnet.mexw32* en la carpeta *bin.*

5.4.4. Invocación del Constructor MEX - Compilación

>> compinterfaz

El constructor MEX es la única función que se localiza directamente en la carpeta CNP3DX. Para ser invocado, esta carpeta debe estar cargada como *current folder* en Matlab. Su invocación se realiza como la de cualquier función .m.

Su código busca en la capeta *src* al archivo C++ y los demás archivos enlazados para compilarlos, y retorna a la carpeta *bin* al archivo binario.

Retorno

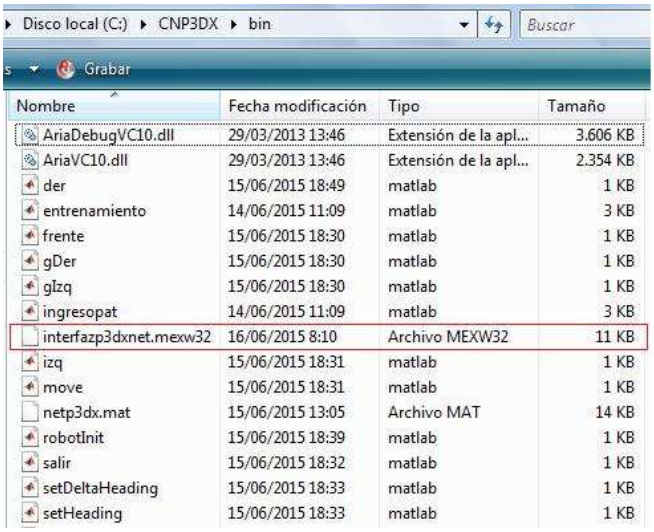


Figura 45. Carpeta bin de la aplicación

El recién creado archivo MEX binario *intp3dxffn.mexw32* es retornado a la carpeta *bin,* porque debe estar junto al resto de archivos de Matlab y a las librerías *AriaVC10.dll* y *AriaDebugVC10.dll* para que la aplicación funcione correctamente.

Se debe recordar que el archivo MEX binario es reconocido por Matlab como una función más, por lo tanto debe estar en la carpeta *bin* con el resto de funciones.

Las librerías dinámicas *AriaVC10.dll* y *AriaDebugVC10.dll* son los módulos que enlazan con Aria, sin ellas la interfaz falla. Deben ser copiadas desde la carpeta *bin* de Aria, path: C:\ProgramFiles\MobileRobots\ARIA\bin.

5.5. Funciones del proyecto

¿Cuántas funciones son de control?,
¿Cuál es la diferencia entre las funciones primitivas y compuestas?,
¿Por qué hay funciones pasarela? ¿Qué significa funciones de configuración, de sensoramiento y de actuación? ¿Son las mismas funciones, que las de las clases de Aria?

En la Figura 46 se muestra un esquema de todas las funciones usadas en el proyecto, y como estas están clasificadas.

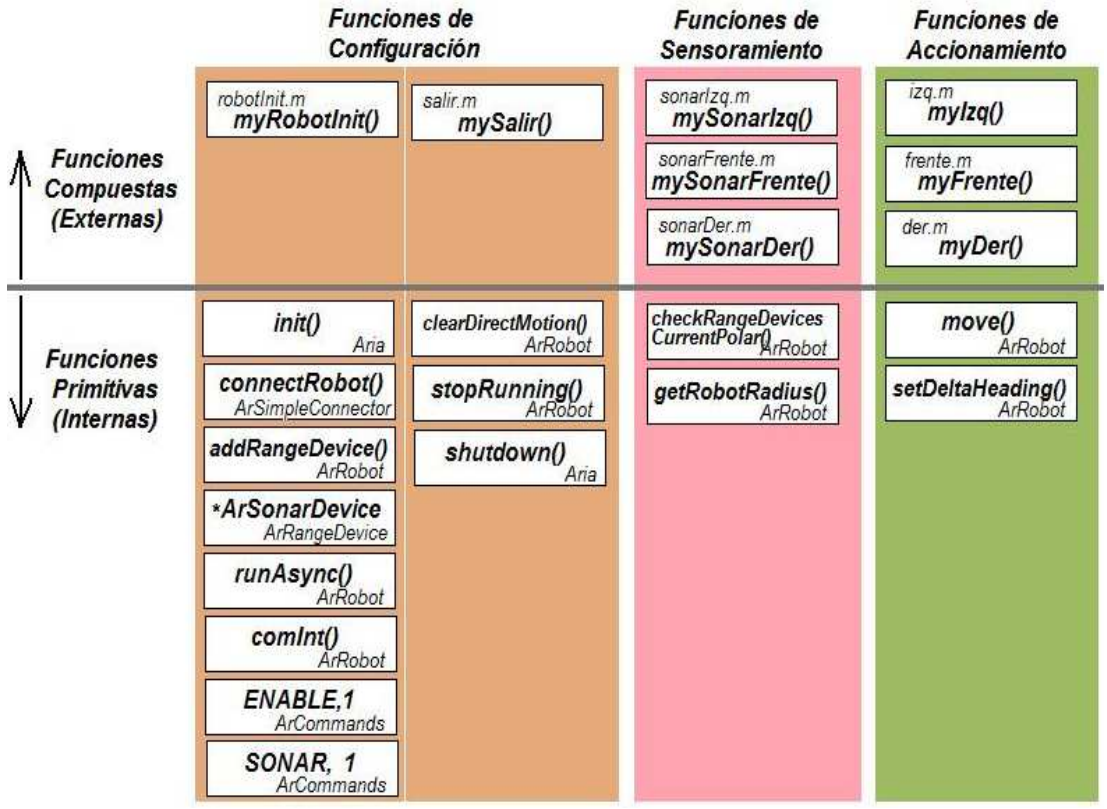


Figura 46. Esquema de funciones del proyecto

5.5.1. Funciones para el control del robot

En total se dispone de ocho funciones útiles para controlar el robot, divididas en tres grupos: de configuración, que sirven para conectar-encender el robot y para detenerlo-desconectarlo; de sensoramiento, que sirven para obtener las lecturas actualizadas del sonar, y de accionamiento, que sirven para mover el robot.

Todas estas funciones están declaradas en la rutina de cálculo del archivo MEX C++ (sección 5.3.1). En Matlab están escritas sus respectivas funciones .m, que sirven para invocarlas a través de la rutina de enlace.

En la Tabla 20 se muestra el flujo de invocación de las funciones de control, desde el algoritmo de control hasta el archivo MEX.

Tabla 20.

Flujo de funciones de control

	Matlab		**Archivo MEX**
	trayectoria.m	***funciones .m*** *(respectivas)*	***interfazp3dxnet.cpp***
FC	robotInit()	interfazp3dxnet(0)	myRobotInit()
FC	salir()	interfazp3dxnet(1)	mySalir()
FS	sonarIzq()	[sensorArray]=interfazp3dxnet(10)	mySonarIzq(output[])
FS	sonarFrente()	[sensorArray]=interfazp3dxnet(11)	mySonarFrente(output)
FS	sonarDer()	[sensorArray]=interfazp3dxnet(12)	mySonarDer(output[])
FA	izq()	interfazp3dxnet(20)	myIzq()
FA	frente()	interfazp3dxnet(21)	myFrente()
FA	der()	interfazp3dxnet(22)	myDer()
Fal	move(dis)	interfazp3dxnet(2,dis)	robot.move()

Nota: la última función es alterna, no es de control, se usa para colocar en una posición inicial al robot.

5.5.2. Funciones Compuestas

Todas las funciones de control son compuestas, porque son programas código fuente que invocan a funciones de las clases de la librería Aria.

Los ejemplos más explícitos son my*RobotInit*() y *mySalir(),* en ambos se accede a un conjunto de funciones para conseguir que el robot o se encienda-y-conecte o se desconecte-y-apague.

Dicho de otra manera, las funciones compuestas pueden ser vistas como “banderas”, que activan un conjunto de funciones de las clases de Aria para conseguir una acción de control.

5.5.3. Funciones Primarias o Primitivas

Todas las funciones de las clases de Aria que han sido utilizadas para armar las funciones compuestas, son llamadas primarias o primitivas Aria.

Siguiendo con el ejemplo de la función compuesta *mySalir()*, a continuación con ayuda de la Figura 47 se analiza cómo fluye a través del archivo MEX C++.

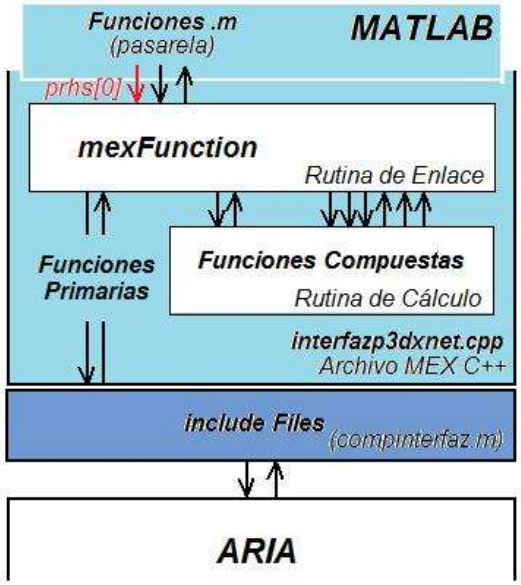


Figura 47. Flujo de funciones a través del archivo MEX

Primero, en Matlab es invocada la función *salir.m.* Esta sirve de pasarela y envía a la *mexFunction* de la rutina de enlace un pedido con *prhs[0]=1*. La rutina de enlace analiza el pedido y activa la función compuesta *mySalir().*

Esta función compuesta invoca tres funciones de Aria para conseguir apagar y desconectar el robot:

- *.clearDirectMotion()* → de la clase ArRobot, para borrar cualquier comando,
- *.stopRunnig()* → de la clase ArRobot, para detener cualquier proceso en el robot, y
- *::shutdown()* → de la clase principal Aria, para cerrar todos los procesos de Aria.

5.5.4. Funciones Comandos de Movimientos de Aria

Aria es una librería precompilada C++, que agrupa todas las *Motion Commands Functions* (y los *Direct Commands*) útiles para controlar la plataforma, a través de un objeto de la clase *ArRobot* (sección 3.13.2).

El archivo de inclusión "*Aria.h"* está declarado en el archivo MEX (sección 5.3.1), además los archivos de inclusión de cada clase han sido enlazados dinámicamente en la compilación del archivo MEX binario (sección 5.4.3), para garantizar el acceso y uso de todas las funciones de Aria.

Se recomienda revisar el código de la función compuesta *myRobotInit()* (sección 5.3.1 – Funciones de configuración). En éste se indica la estructura que debe tener un programa de control Aria, partiendo desde la generación de los objetos para la comunicación.

5.5.5. Funciones .m pasarela (de enlace)

Estas funciones de Matlab, como bien dice su nombre, sirven para enlazar las funciones de control, cuando son invocadas en el algoritmo de control del robot *trayectoria,* con sus respectivas funciones en el archivo MEX.

A continuación se presenta el código de tres de estas funciones:

```
// sonarIzq.m
function [sensorArray] = sonarIzq()
error(nargchk(0, 0, nargin))
sensorArray = interfazp3dxnet(10);

// der.m
function der()
error(nargchk(0, 0, nargin))
interfazp3dxnet(22);

// move.m
function move(distance)
error(nargchk(1, 1, nargin))
interfazp3dxnet(2,distance);
```

Se han generado funciones pasarela a cada función compuesta, Tabla 20, y también a algunas funciones primitivas directamente, como por ejemplo move.m.

A decir verdad, toda función Aria que este cargada al objeto robot de *ArRobot* y por ende a la conexión, puede ser directamente invocada desde el algoritmo de control, con una función paralela declarada en el selector de la rutina de acceso.

Funciones alternas

Tabla 21.

Funciones alternas – modo manual

	Matlab		Archivo MEX
	trayectoria.m	***funciones .m*** *(respectivas)*	***interfazp3dxnet.cpp***
Fal	move(dis)	interfazp3dxnet(2,dis)	robot.move(dis)
Fal	setDelta Heading(dgr)	interfazp3dxnet(3,dgr)	robot.setDeltaHeading(dgr)
Fal	setHeading (dgr)	interfazp3dxnet(4,gra)	robot.setHeading(gra)
Fal	setVel(mms)	interfazp3dxnet(5,mms)	robot.setVel(mms)
Fal	setRotVel(gs)	interfazp3dxnet(6,gs)	robot.setRotVel(gs)
Fal	setVel(mms)	[vel]=interfazp3dxnet(7)	robot.setVel(mms)
Fal	setRotVel(gs)	[rVel]=interfazp3dxnet(8)	robot.setRotVel(gs)
Ca	gIzq()	interfazp3dxnet(23)	myGIzq() → robot.setDeltaHeading(30)
Ca	gDer()	interfazp3dxnet(24)	myGDer() → robot.setDeltaHeading(-30)

En este proyecto se tiene a disposición siete funciones pasarela directas

a las funciones de la clase ArRobot, que se emplean aisladamente para mover y configurar al robot sin tener que acceder al algoritmo de control de trayectoria.

Basta con que el robot este encendido y conectado para operarlo de manera “manual” con las funciones alternas de la Tabla 21.

Nota: las funciones compuestas alternas *gIzq()* y *gDer()* generan giros fijos de 30° que solo se usan para mover al robot a una posición inicial específica, en modo manual. Fueron desarrollados como una versión abreviada de setDeltaHeading(+/- 30°).

Recuerde, las funciones de las clases de Aria cargadas al objeto de ArRobot pueden ser llamadas directamente con una función pasarela, a través de un caso en el selector de la rutina de enlace. Sin tener que pasar por una función compuesta.

5.6. Objetos C++ de Aria del proyecto

En la Figura 48 se muestra un diagrama de bloques que junta a todos los objetos instanciados en la rutina de cálculo del archivo MEX C++, y que interactúan con el objeto que se genera en *el Neural Network Toolbox.*

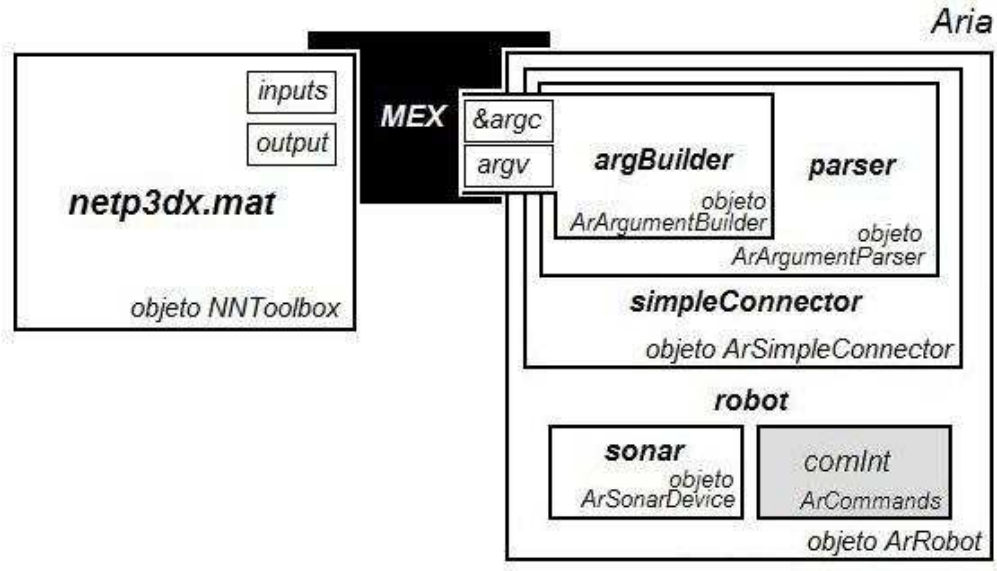


Figura 48. Objetos Aria y NN Matlab del proyecto

5.7. Composición de la carpeta CNP3DX

La carpeta CNP3DX contiene todos los archivos necesarios para la correcta ejecución de la aplicación. Si no se han hecho las actualizaciones respectivas de los *paths* en el constructor MEX (sección 5.4.2), no se debe alterar su orden.

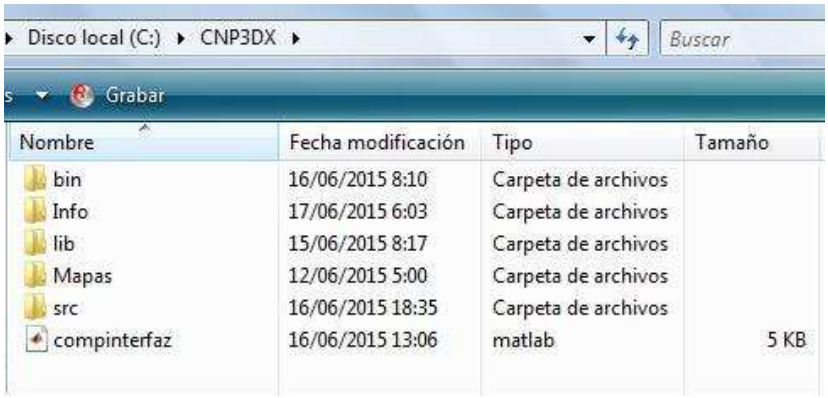


Figura 49. Carpeta CNP3DX

El constructor MEX *compinterfaz.m* está colocado en esta carpeta solo. Si es reubicado, se deben actualizar los directorios de fuente y de destino de la función *mex* en el constructor (sección 5.4.3).

Recuerde que Matlab reconoce las funciones que se encuentran cargadas en su *'current folder'*. Por lo tanto para compilar, fijarse que esta sea la carpeta actual de Matlab.

5.7.1. Carpeta bin

Después de haber compilado el binario, para que el proyecto pueda ser ejecutado en Matlab, en el *current folder* debe estar abierta la carpeta bin.

De esta manera se garantiza que el path del archivo MEX binario es reconocido por el Matlab.

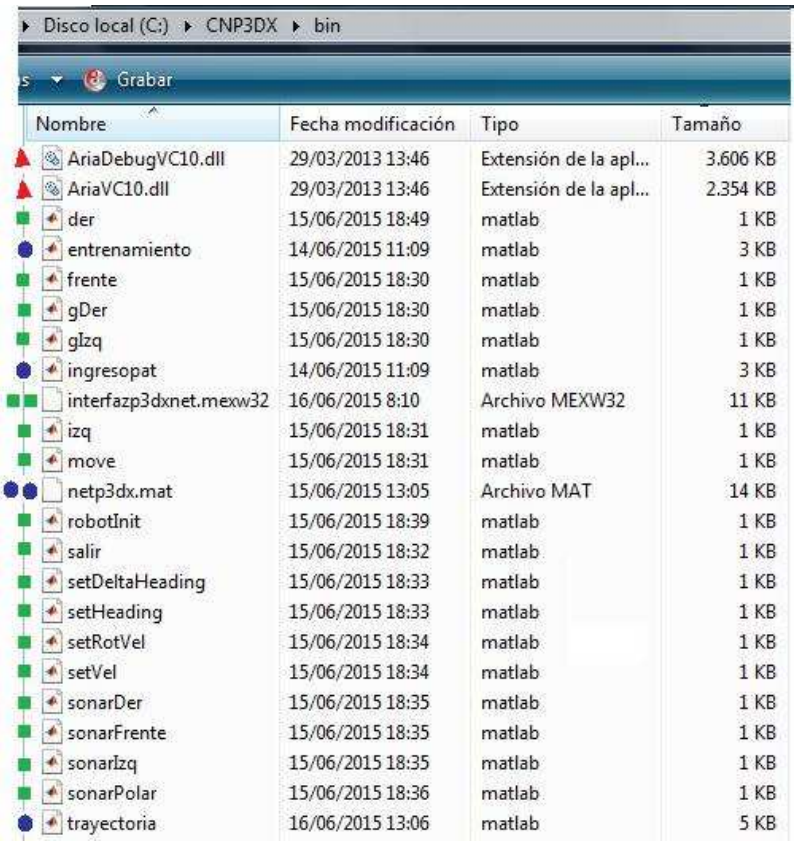


Figura 50. Carpeta bin

En esta carpeta se recogen archivos de tres fuentes:

- en color verde todos los archivos que permiten la ejecución de la interfaz Matlab-Aria,
- en azul todos los que pertenecen al neurocontrolador, y
- en rojo las librerías dinámicas de Aria.

Nótese que en esta carpeta se ubica el archivo MEX binario *interfazp3dxnet.mexw32* acabado de compilar y el neurocontrolador entrenado *netp3dx.mat*.

5.7.2. Carpeta lib – libMV2003

En esta carpeta se guardan las librerías estáticas importadas desde Microsoft Visual Studio .NET 2003.

Figura 51. Carpeta lib

5.7.3. Carpeta Mapas

Para probar el comportamiento del robot se han desarrollado tres mapas en el simulador MobileSim.

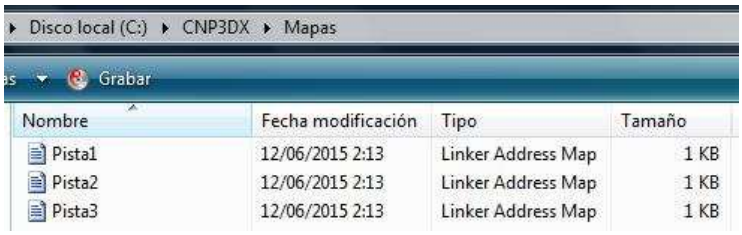


Figura 52 Carpeta de mapa

5.7.4. Carpeta src

En esta carpeta se localiza el archivo MEX código fuente. Si hubiese más archivos C++ para ser combinados en la compilación, se recomienda colocarlos a todos aquí.

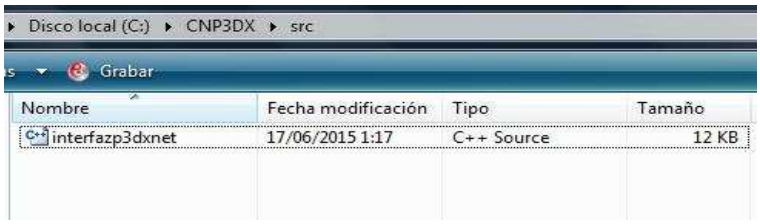


Figura 53. Carpeta src

6. DESARROLLO DE LA APLICACIÓN

6.1. Planteamiento del Problema

Inspirado en la aplicación introductoria que David Kriesel publicó en su libro "A Brief Introduction to Neural Network".

6.1.1. Problema

Controlar la trayectoria de la plataforma robótica móvil Pioneer P3-DX, cuando ésta se mueve dentro de una pista para evitar colisiones con las paredes. La plataforma no tiene una posición inicial definida dentro de la pista, ni tampoco un sentido de giro definido.

6.1.2. Solución

La plataforma Pioneer P3-DX es un robot compacto, que tiene un arreglo frontal de sensores ultrasónicos (sección 1.7.2) para extraer datos de entrada. Estos datos son valores numéricos reales, que pueden ser retornados en cualquier momento, en intervalos de 100 milisegundos.

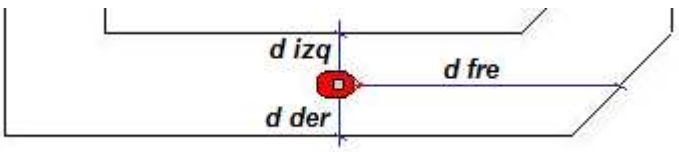


Figura 54. Distancia de sensoramiento

Como el robot se desplaza encerrado por paredes dentro de una pista

(Figura 54), es necesario saber a qué distancia lateral están dichas paredes y a qué distancia frontal se avecina una curva.

Esto significa que permanentemente se reciben tres valores numéricos reales de entrada, correspondientes a la distancia que hay con la pared a la izquierda, al frente y a la derecha: $inputs \in R^3$.

El robot debe moverse hacia delante, evitando chocar con las paredes. La salida es un valor numérico real que representa una orden a los motores actuadores, para que generen el desplazamiento: $outputs \in R^1$.

Entonces, la solución de este problema se reduce a la aproximación de la función no lineal $f: R^3 \rightarrow R^1$, que aplica las señales de entrada a una activación del robot.

A esta función se la pueda aproximar de la manera clásica o mediante aprendizaje. En la manera clásica, la función puede ser resuelta con algún operador lógico, siempre que sea simple. Si se torna compleja, de un programa pequeño de control se puede pasar a uno extenso, rígido y lento de computar.

Por el contrario, el aprendizaje es una alternativa más interesante y exitosa para aproximaciones de funciones no lineales complejas y difíciles de comprender. El robot aprende de determinados escenarios ejemplos (sección 6.2.1), y cuando está en marcha generaliza lo aprendido mostrando un comportamiento inteligente y autónomo.

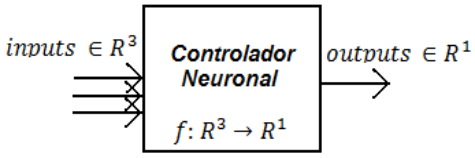


Figura 55. Aproximación de función con neurocontrolador
Fuente: (Kriesel, 2007)

Precisamente la capacidad de aprender es una de las características

primarias de las redes neuronales artificiales (Figura 55). Por lo tanto, un controlador inteligente diseñado con una red neuronal, es la solución precisa para obtener el comportamiento deseado en el robot.

6.2. Desarrollo de Neurocontrolador

Inspirado en el proceso de diseño propuesto por Beale, Hagan y Demuth en la guía de usuario del Neural Network Toolbox. (Sección 3.1)

El neurocontrolador es desarrollado con una red neuronal multicapa alimentada hacia delante, entrenada con un algoritmo que aproxima el cálculo del gradiente de segundo orden. El esquema de control empleado es de tipo adaptativo directo basado en un modelo del entorno, simulado en MobileSim (sección 2.9.1).

Modelo Simulado

En el MobileSim se ha simulado la pista donde el robot se desplaza evitando chocar con las paredes. En esta misma pista es entrenado el neurocontrolador.

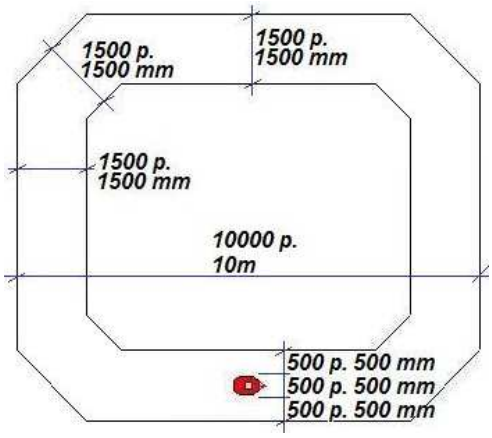


Figura 56. Modelo Simulado - Pista

La resolución del MobileSim indica que cada pixel (punto simulado) representa un milímetro. De tal manera que la pista y el robot son simulados

con las siguientes dimensiones:

- Pista en recta: mide de ancho 1.5 metros (simulados con 1500 pixeles), y de largo máximo 10 metros.
- Pista en giro de 45°: mide 1.5 metros de ancho y un máximo de 4.5 metros de largo.
- Pioneer P3-DX: es simulado con 500 pixeles de ancho, es decir 0.5 metros. Si se considera que el robot físico tiene un radio de giro interno de 26.7 centímetros (sección 1.7.3), un diámetro simulado de 50 centímetros cubre ajustadamente al robot en un giro.

Si se pudiese simular en tres dimensiones en MobileSim, la altura de las paredes debería ser mayor a 24 centímetros, que es la altura del robot, para garantizar que sean reconocidas por los sensores.

6.2.1. Recolección de patrones desde el modelo

Los autores de la guía de usuario del Toolbox advierten que:

> generalmente es difícil incorporar conocimiento previo en una red neuronal, por lo tanto la red puede ser solo tan precisa como los datos que son usados en su entrenamiento. La red no es capaz de extrapolar con presión más allá del rango de entradas, por eso es importante que los patrones de entrenamiento cubran completamente dicho rango.

Si los patrones han sido correctamente seleccionados, la red neuronal generaliza desde los ejemplos y encuentra una regla universal para evitar choques con las paredes. (Kriesel, 2007)

Composición de los patrones de entrenamiento

Las redes alimentadas hacia adelante *'feedforwardnet'* son entrenadas en el *Toolbox* en aprendizaje supervisado. Por lo tanto, los patrones de entrenamiento van a estar compuestos por cuatro valores numéricos reales:

tres de entradas (Tabla 22) y uno de objetivo *target* (Tabla 23).

Las *entradas* varían desde un valor mínimo de cero a un máximo de 4750, correspondientes a los 5000 milímetros de alcance máximo del arreglo de sensores menos 250 del radio interno del robot. Para el entrenamiento las entradas son normalizadas en el rango $[0, +4.75]$.

Tabla 22.

Entradas del neurocontrolador

Entradas *Inputs*	**Correspondencia**	**Características** Región barrida (Figura 57)
in1	Lectura del sensor izquierdo	De 90° a 70°
in2	Lectura del sensor frontal	De 10° a -10°
in3	Lectura del sensor derecho	De -70° a -90°

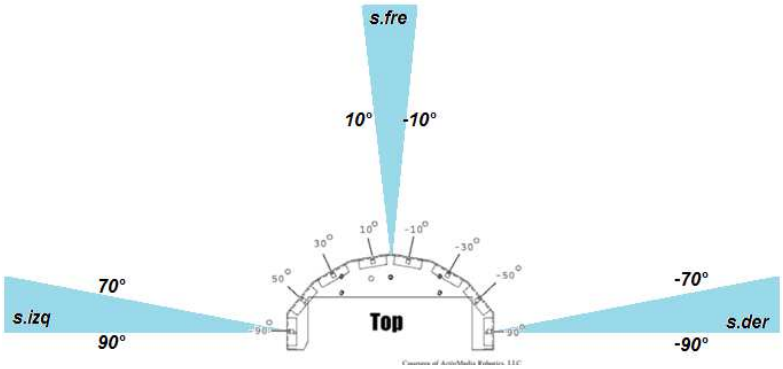


Figura 57. Regiones de sensoramiento en la aplicación

Fuente: (MobileRobots - ActivMedia Robotics, 2006)

Como respuesta a los tres valores sensados de entrada, el neurocontrolador dispone tres posibles acciones de movimiento, representadas por un valor numérico real fijo –*objetivo*-, ver Tabla 23.

Tabla 23.

Targets del neurocontrolador

Acciones	**Referencia** *Targets*	**Movimiento** (Figura 58)
Movimiento 1: ***Giro izquierda y avance***	-1	Robot gira a la izquierda 30° y avanza 200 milímetros
Movimiento 2: ***Solo avance***	0	Robot avanza 350 milímetros
Movimiento 3: ***Giro derecha y avance***	1	Robot gira a la derecha 30° y avanza 200 milímetros

Figura 58. Movimientos del P3-DX en la aplicación

Lógica para la adquisición de patrones

Cuando el robot se desplaza por la pista puede hacerlo por diferentes trayectorias y ubicándose en incontables posiciones. Registrar todas las posibles posiciones y definir la acción a realizar en cada una, sería una tarea excesivamente larga y monótona en la que seguramente se producirían fallos, como relaciones entradas-salidas repetidas o contradictorias.

Para el entrenamiento han sido registradas posiciones ejemplo con sus respectivos movimientos de respuesta. Para que estas sean suficientemente representativas y el robot no se choque con las paredes, la pista ha sido dividida en tres regiones, dos de alarma y una central (mirar Figura 59).

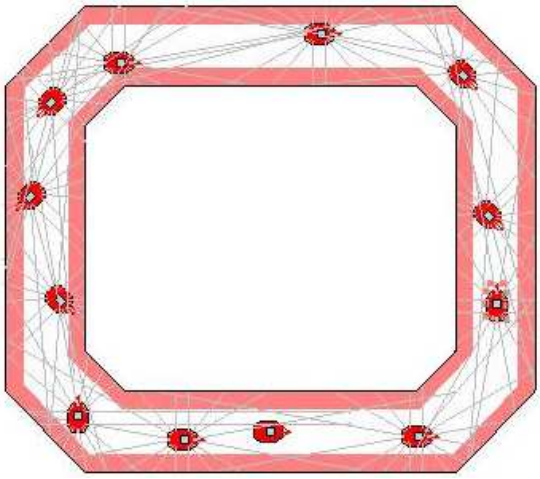


Figura 59. Recolección de patrones en el modelo

Las regiones de alarma son las de los extremos y exigen un giro del

robot porque un choque es inminente. En la región central se le permite al robot avanzar de frente, pues está a una distancia prudente de las paredes.

El ancho aproximado de las regiones de alarma es de 300 milímetros, que cubren a los 267 milímetros del radio interno de giro del robot (sección 1.7.3). Es decir, si el robot ya estuviese dentro de la zona de alarma este puede girar en la dirección más inteligente, sin golpearse contra las paredes.

Si los valores de las lecturas de los sensores de la izquierda *in1* y del frente *in2* se aproximan o ya son menores a 300, significa que el robot está muy cerca de la pared de la izquierda. La respuesta más inteligente del neurocontrolador es hacerle girar a la derecha al robot y que avance (*targets=1* → movimiento 3 en Tabla 23). Este ha sido un ejemplo de la lógica con la que se han recolectado los patrones de entrenamiento.

El robot ha sido colocado en distintas posiciones, sobre y/o entre las tres regiones de la pista, para conseguir valores de entrada desde los sensores. A estos se les ha añadido el valor fijo deseado de salida *targets,* conforme la lógica de recolección, quedando así armados los patrones de entrenamiento.

Ingreso de Patrones – Código de Matlab

Ubicación del programa: CNP3DX \ bin\ingresopat.m

```
% Descripción:
% Mediante este programa función los patrones de entrenamiento son
% recolectados y correctamente ubicados en el arreglo.

function [inputs,targets]=ingresopat

P=[];
% Lista de Patrones de Entrenamiento
% ----------------------------------
% 3 entradas (sensor izq, s.fre, s.der) y 1 target
p{1}=[0.49 4.02 0.49  0];
p{2}=[0.89 0.72 0.33 -1];
p{3}=[0.33 0.72 0.91  1];
p{q}=[ in1  in2  in3  t];
% Formación matriz de patrones P=[]Qx(R+1)
% ---------------------------
For k = 1:size(p,2)
    P = [P;p{k}];
% Matriz transpuesta de patrones PT=[](R+1)xQ
PT=transpose(P);
```

```
% Separación de arreglo de entradas, y targets
% ---------------------------------------------
% % Referencia: sección 3.3.2 - patrones de entrenamiento
% Arreglo de entradas IN=[]RxQ
IN=PT(1:3,:);
% Targets TARG=[]1xQ
TARG=PT(4,:);

% Asignación de variables a función
% ---------------------------------
inputs=IN;
targets=TARG;
```

El arreglo completo de patrones de entrenamiento se encuentra directamente en el archivo del programa adjunto a este informe.

6.2.2. Creación del neurocontrolador

Ubicación del programa: CNP3DX \ bin\entrenamiento.m

```
% Descripción:
% En este programa se crea la red multicapa alimentada hacia
% adelante, y luego se la entrena. Antes del entrenamiento la red
% es configurada, y los pesos y bias son reinicializados

clear all
clc

% Ingreso de patrones de entrenamiento
% -------------------------------------
% Desde la función ingresopat.m
[inputs,targets]=ingresopat

% Creación de red
% ---------------
% % Referencia: sección 3.2.3 feedforwardnet
% net=feedforwardnet(hiddenSizes,trainFcn)
netp3dx=feedforwardnet(36);
% feedforwardnet o fitnet (feedforwardnet especializada para
% aproximación, mismos resultados)
% trainFcn = 'trainlm'-> precargado, 'trainbfg' y 'trainoss'
```

Cálculo de número de neuronas en la capa escondida

Para tener un error cuadrático medio menor a 0.02777 se necesitan 36 neuronas en la capa oculta. Referencia: sección 2.7.1, ecuación (25)

$$E = \mathcal{O}\left(\frac{1}{h}\right)$$

$$E = \mathcal{O}\left(\frac{1}{36}\right) \cong 0.02777$$

Objeto netp3dx retornado (referencia: sección 3.2.4)

```
dimensions:                 functions:

      numInputs: 1                adaptFcn: 'adaptwb'
      numLayers: 2               adaptParam: (none)
     numOutputs: 1                 derivFcn: 'defaultderiv'
  numInputDelays: 0                divideFcn: 'dividerand'
  numLayerDelays: 0              divideParam: .trainRatio, .valRatio, .testRatio
numFeedbackDelays: 0               divideMode: 'sample'
numWeightElements: 36                 initFcn: 'initlay'
      sampleTime: 1                 performFcn: 'mse'
                                  performParam: .regularization, .normalization, .squaredWeighting
  connections:                        plotFcns: {'plotperform', plottrainstate, ploterrhist,
                                                 plotregression}
     biasConnect: [1; 1]            plotParams: {1x4 cell array of 1 param}
    inputConnect: [1; 0]              trainFcn: 'trainlm'
    layerConnect: [0 0; 1 0]        trainParam: .showWindow, .showCommandLine, .show, .epochs,
   outputConnect: [0 1]                         .time, .goal, .min_grad, .max_fail, .mu, .mu_dec,
                                                .mu_inc, .mu_max
```

Figura 60. Objeto netp3dx creado

Funciones preseleccionadas:

- de entrenamiento → *'trainlm',*
- de rendimiento → *'mse'* error cuadrático medio, y
- de división → 'dividerand' aleatoria.

Subobjetos netp3dx retornados

>>netp3dx.layers{1} → capa oculta

>>netp3dx.layers{2} → capa de salida

```
          name: 'Hidden'                    name: 'Output'
    dimensions: 36                    dimensions: 1
   distanceFcn: (none)               distanceFcn: (none)
 distanceParam: (none)             distanceParam: (none)
     distances: []                     distances: []
        initFcn: 'initnw'                 initFcn: 'initnw'
   netInputFcn: 'netsum'             netInputFcn: 'netsum'
 netInputParam: (none)             netInputParam: (none)
     positions: []                     positions: []
         range: [36x2 double]              range: [1x2 double]
          size: 36                          size: 1
  topologyFcn: (none)               topologyFcn: (none)
  transferFcn: 'tansig'             transferFcn: 'purelin'
transferParam: (none)             transferParam: (none)
      userdata: (your custom info)      userdata: (your custom info)
```

Figura 61. Subobjetos netp3dx creados

La combinación de funciones de transferencia *'tansig'/'purelin'* es la idónea para una aproximación. Los valores fijos del *targets* son -1, 0 y 1, cada uno representa una acción de control, y están perfectamente cubiertos

por la salida de ‘*tansig’* (sección 3.2.2).

6.2.3. Entrenamiento del neurocontrolador

Ubicación del programa: CNP3DX \ bin\entrenamiento.m

La creación y el entrenamiento del neurocontrolador son realizados en el mismo programa de Matlab.

```
% Configuración e Inicialización
% ------------------------------
% Referencia: sección 3.3.1

netp3dx=configure(netp3dx,inputs,targets);
netp3dx=init(netp3dx);

% División de patrones, antes del entrenamiento
% ---------------------------------------------
% % Referencia: sección 3.4.1 - División de patrones
% netp3dx.divideParam.trainRatio=80/100;
% netp3dx.divideParam.valRatio=10/100;
% netp3dx.divideParam.testRatio=10/100;
% % Referencia: sección 3.4.2 - Entrenamiento sin validación/prueba

netp3dx.divideFcn='dividetrain';

% Entrenamiento de red
% --------------------
% % Referencia: sección 3.3.2
% Algoritmos de entrenamiento rápido comparados
% netp3dx.trainFcn='trainbfg'
% netp3dx.trainFcn='trainoss'

[netp3dx,tr]=train(netp3dx,inputs,targets);

% Prueba de red (visualización de resultados de entrenamiento)
% -------------
% % Referencia: sección 3.5 - Uso de red
% % Referencia: sección 2.4.2 - Función de transferencia lineal
% Retorna las salidas puras del entrenamiento, son valores continuos
% 'purelin'

outputs=netp3dx(inputs)

% Los salidas son redondeadas para que los valores se discreticen y
% coincidan con los targets -1 0 1

outputs_redondeadas=round(outputs)

% % % Simulación alternativa para evaluar red
% % Referencia: sección 3.5.1
% outputs=sim(netp3dx,inputs);

% ATENCION: Se guarda la red entrenada netp3dx.mat para ser
% utilizada en el algoritmo de control de trayectoria
save netp3dx
```

Todos los parámetros que direccionan el entrenamiento deben ser modificados siempre antes de que este ocurra.

Ventana auxiliar de entrenamiento retornada

El parámetro que corto el entrenamiento fue la gradiente. El valor mínimo conseguido es menor al límite inferior *min_grad* (sección 3.3.4 – Patrones de entrenamiento). Los resultados son analizados en el capítulo 7, sin embargo brevemente se puede decir que el entrenamiento es bueno por los valores conseguidos del error cuadrático medio y de la gradiente.

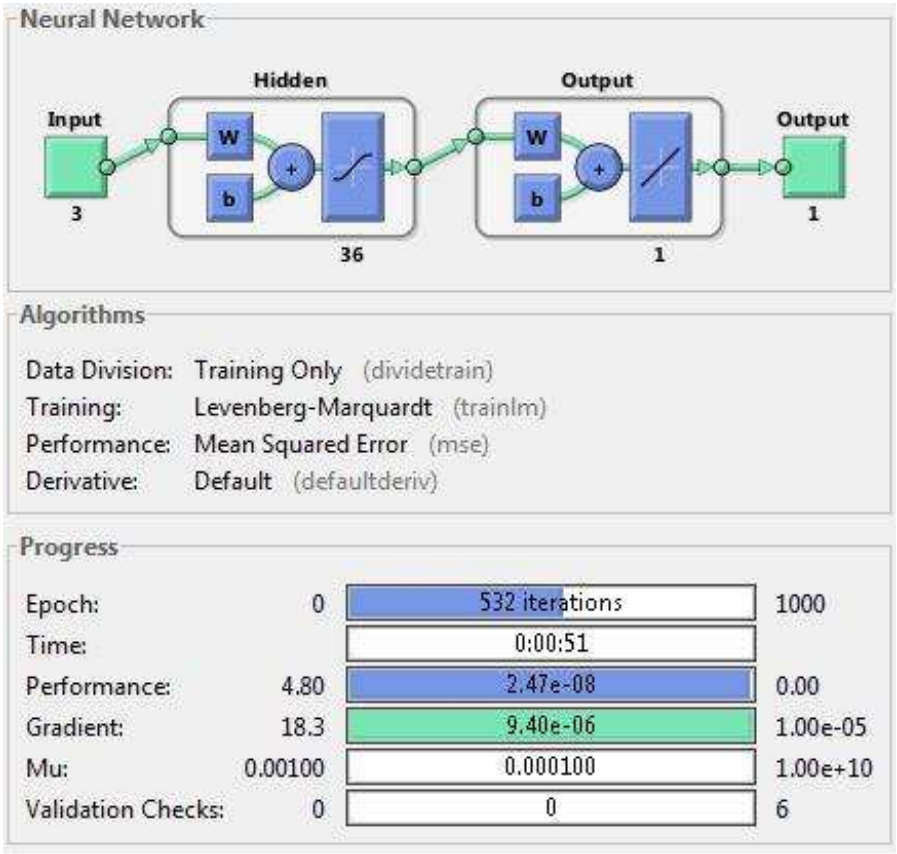


Figura 62. Ventana auxiliar de entrenamiento en aplicación

Valor retornados a la ventana de Comandos

En la Tabla 24 se muestra una fracción de las salidas obtenidas en el entrenamiento. Estas coinciden con sus respectivos targets. Nuevamente se puede decir, que es un indicador positivo de que la red entrenada está aproximando bien.

Tabla 24.

Valores numéricos de entrenamiento en aplicación

Elemento	*Valores numéricos reales*
Entradas entrena-miento	inputs = Columns 1 through 10 0.4900 0.3400 0.6300 0.6500 0.8900 0.3300 0.9600 0.5500 0.7200 0.7300 4.0200 3.0700 2.6500 0.9600 0.7200 0.7200 2.2600 2.2600 1.8700 1.5200 0.4900 0.6500 0.3300 0.3300 0.3300 0.9100 0.5500 0.9600 0.4500 0.3400
Targets	targets = Columns 1 through 17 0 0 0 0 -1 1 0 0 0 0 0 0 1 1 1 -1
Salidas de entrena-miento	outputs_redondeadas = Columns 1 through 16 0 0 0 0 -1 1 0 0 0 0 0 0 1 1 1 -1

6.2.4. Uso del neurocontrolador

Ubicación del programa: CNP3DX \ bin\trayectoria.m

El neurocontrolador ha sido diseñado bajo esquema adaptativo directo, es decir las señales de sensamiento (las tres lecturas del SONAR) y la señal de control (el output de la red) son recibidos y enviado directamente entre el Pioneer P3-DX y la red (sección 2.9.1).

```
% Descripción:
% En este programa se realiza el control de trayectoria
% del Pioneer P3-DX mediante el Uso de la Red. La transferencia de
% las señales entre el robot y el neurocontrolador es directa

% Referencia de Funciones pasarela a archivos MEX: sección 5.5.5

clear all
clc

% Etiqueta introductoria:
% Título retornado a la ventana de comandos cuando se corre el
% programa de control.

% Red cargada
% ---------------
% Para ser usada en el algoritmo de control
load netp3dx

% Arranque Robot (motores y sensores) y config. iniciales
% ----------------------------------------------------
fprintf('\nPulsar cualquier tecla para iniciar el control...\n');
pause()
RobotInit() % /// **F.pasarela Archivo MEX 0** ///

% Si se desea setear la velocidad de traslación y rotación
% setVel() % /// **F.pasarela Archivo MEX 5** ///
% setRotVel() % /// **F.pasarela Archivo MEX 6** ///

% Bucle para iteraciones de Operación
while i<=100
    i=i+1;
    % Señalizador de inicio de iteración
```

```
    fprintf('\n(%d) Lectura actual del SONAR\n',i);
    pause(1)

    % Lectura de Sonares - recepción directa
    % ---------------------------------------
    iniz=sonarIzq(); % /// **F.pasarela Archivo MEX 10** ///
    iniz=iniz*0.001; % normalización entrada a rango [0,+4.75]
    infr=sonarFrente(); % /// **F.pasarela Archivo MEX 11** ///
    infr=infr*0.001;
    inde=sonarDer(); % /// **F.pasarela Archivo MEX 12** ///
    inde=inde*0.001;

    % Composición de vector de entradas []Rx1
    inputs=[iniz;infr;inde]

    % Validación de lecturas de los sensores:
    if iniz<0.050 || infr<0.050 || inde<0.050
        % El robot se detiene cuando un choque es inminente
        fprintf('Robot a punto de chocar!...\n');
        fprintf('Detención de emergencia\n');
        fprintf('==============================\n');
        i=i+100;
        salir() % /// **F.pasarela Archivo MEX 1** ///
    end

    % Uso de red (algoritmo de control)
    % ---------------------------------
    % Referencia: sección 3.5
    output=netp3dx(inputs);
    output=round(output)

    % Validación de la salida del entrenamiento
    if output<-1 || output>1
        % si las outputs son menores a -1 o mayores a +1
        % el entrenamiento ha fallado
        fprintf('Entrenamiento deficiente...\n');
        fprintf('Control detenido!\n');
        fprintf('==============================\n');
        i=i+100;
        salir() % /// **F.pasarela Archivo MEX 1** ///
    end

    % Envio directo de la señal de control
    % ------------------------------------

    % Movimiento 1: giro izquierda y avance
    if output==-1 % salida de red = -1
        fprintf('Giro izquierda y avance\n');
        izq(); % /// **F.pasarela Archivo MEX 20** ///
        % Señalizador de fin de iteración
        fprintf('------------------------------\n');

    % Movimiento 2: avance al frente
    elseif output==0 % salida de red = 0
        fprintf('Avance al frente\n');
        frente(); % /// **F.pasarela Archivo MEX 21** ///
        fprintf('------------------------------\n');

    % Movimiento 3: giro derecha y avance
    elseif output==1 % salida de red =+1
        fprintf('Giro derecha y avance\n');
        der(); % /// **F.pasarela Archivo MEX 22** ///
        fprintf('------------------------------\n');
    end
    pause(2)
end
salir() % /// **F.pasarela Archivo MEX 1** ///
```

Retornos durante algoritmo de control

Para verificar que el control inteligente de la trayectoria se esta realizando, se disponen de dos retornos (Figura 63). A la izquierda se muestran dos ejemplos de los retornos constantes en la ventana de comandos de Matlab y a la derecha se muestra la aplicación corriendo en la ventana de MobileSim.

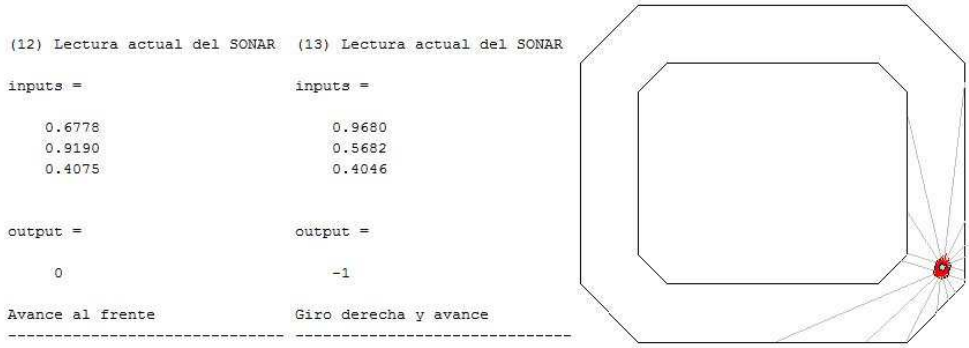


Figura 63. Retorno del control de trayectoria

Cada retorno en Matlab registra los valores de la lectura actual de los sensores *inputs* y la salida evaluada por la red entrenada *output*. Si se pierde el control también se registran los motivos en el retorno, de acuerdo a las dos validaciones descritas en el código (Tabla 25).

6.3. Desarrollo del Interfaz plataforma - red

En el *uso del neurocontrolador*, que es el último paso, se ha llegado a invocar a las funciones pasarela a archivos MEX. Una vez que estas han sido invocadas, los datos empiezan a fluir dentro del interfaz desde el programa de Matlab hasta el archivo MEX, para después seguir viajando hasta el firmware del P3-DX.

Todo el flujo de datos que se da a partir de estas funciones, han sido estudiadas detalladamente en el capítulo 5.

7. RESULTADOS

7.1. Presentación de resultados

La presentación de resultados se realiza directamente en la ventana de comandos de Matlab, en conjunto con la ventana de simulación del MobileSim, en el mismo momento que se esta ejecutando la aplicación.

Para correr la aplicación se debe tener abierto el simulador con el mapa cargado. Además se debe verificar que la carpeta actual de Matlab corresponde a la *bin* del proyecto.

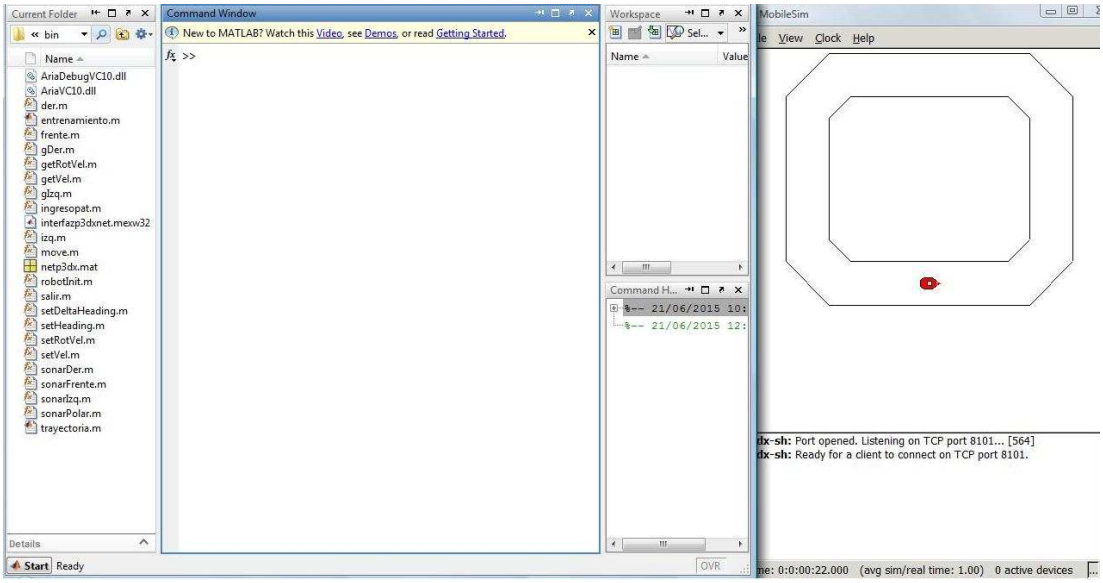


Figura 64. Previas al arranque

Desde el simulador se tiene un retorno, indicando que el puerto del robot está abierto y esperando cualquier conexión de un cliente en el puerto TCP del ordenador.

7.1.1. Arranque de la aplicación

Invocación: >>*trayectoria*

Para arrancar la aplicación se debe invocar al algoritmo de control neuronal, llamado trayectoria.

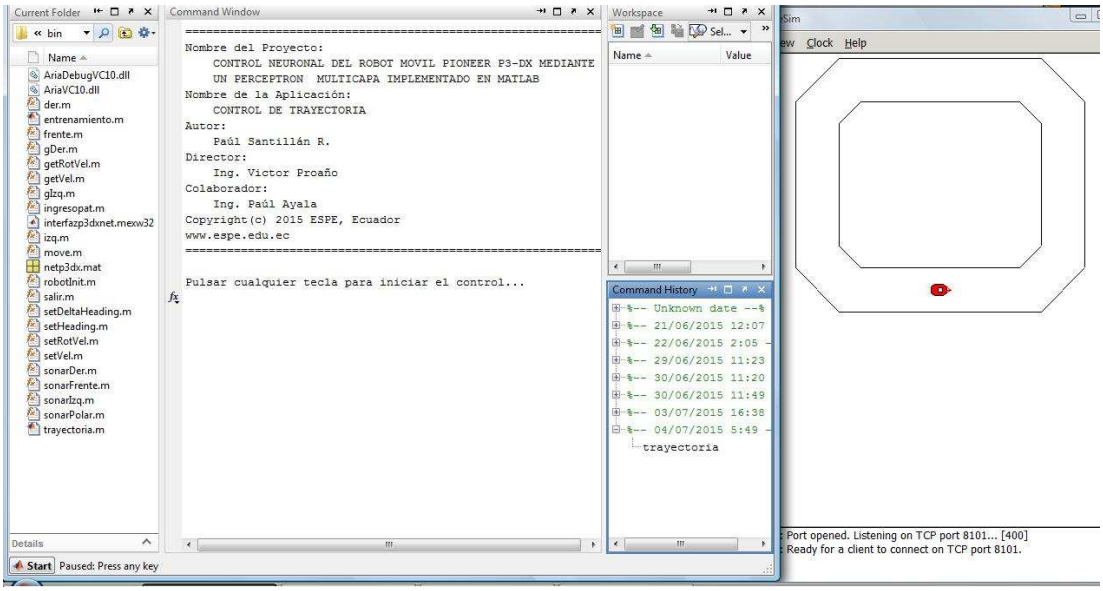


Figura 65. Arranque de la aplicación

Inmediatamente se devuelve una plantilla de presentación, y la orden para iniciar el control.

Una vez iniciado, se conecta y se enciende el robot. Esto se visualmente diferenciable. En el simulador se genera un campo de líneas alrededor del robot, que representan a la región de barrido de los sensores ultrasónicos.

Arranque modo manual

En el modo manual se configura la posición inicial del robot en la pista, con las funciones que en el algoritmo de control son invocadas automáticamente y otras dispuestas para este modo (Tabla 20 y Tabla 21). Para arrancar en modo manual, en lugar de invocar a *trayectoria*, se invoca directamente la función *robotInit(), y* para apagarlo a la función *salir().*

7.1.2. Aplicación en ejecución

La aplicación de control se ejecuta en ciclo retroalimentado, que se muestra en la Figura 66.

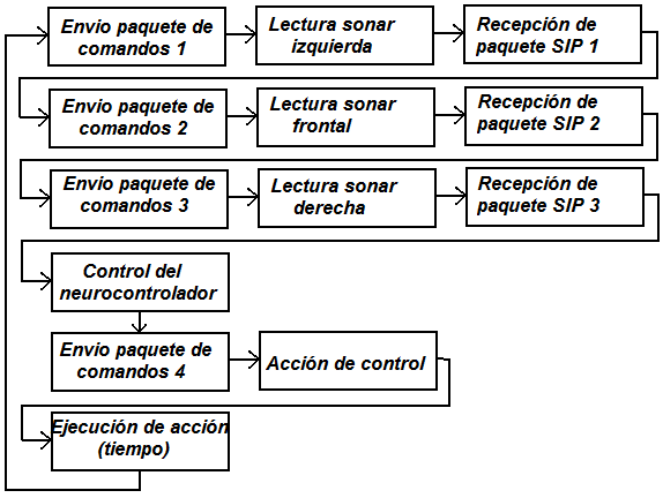


Figura 66. Ciclo de ejecución

Cada ciclo de ejecución es mostrado en la ventana de comandos, numerado solo por referencia. Con *inputs* se registra las lecturas actualizadas del sonar: izquierdo, frontal y derecho, con *output* la salida generada por el neurocontrolador respecto de las entradas del sonar.

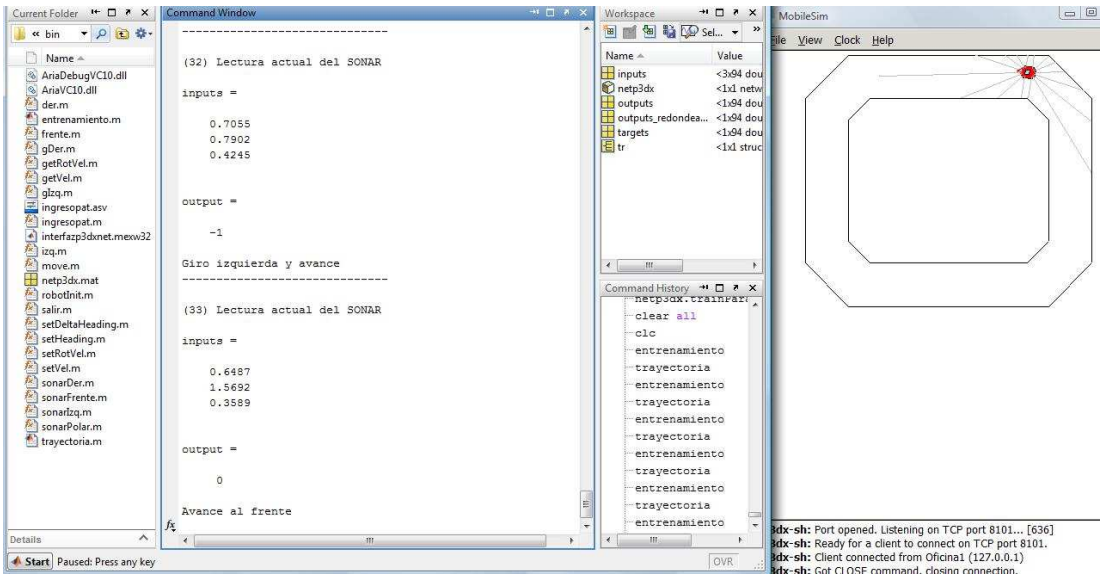


Figura 67. Aplicación en ejecución

Salidas de Emergencia

Se pueden producir dos tipos de salida de emergencia, una por choque inminente y otra por mal entrenamiento.

Tabla 25.

Salidas de emergencia de la ejecución

Tipo de salida	Criterio	Ejemplo
Mal entrenamiento	Cuando el neurocontrolador genera una salida no acotada, a pesar de que las entradas están dentro del rango de entrenamiento	Figura 68
Choque inminente	Cuando una o varias lecturas del sonar muestran que el robot está demasiado cerca de una pared y un choque es inminente. Este tipo de salida es también producto de un mal entrenamiento, puntualmente de una mala generalización.	Figura 69

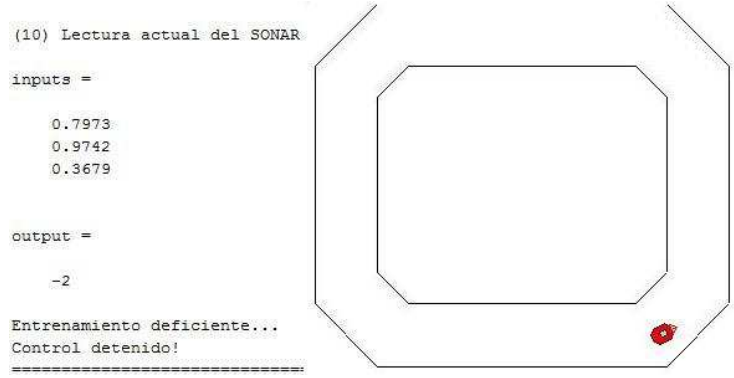


Figura 68. Error, mal entrenamiento

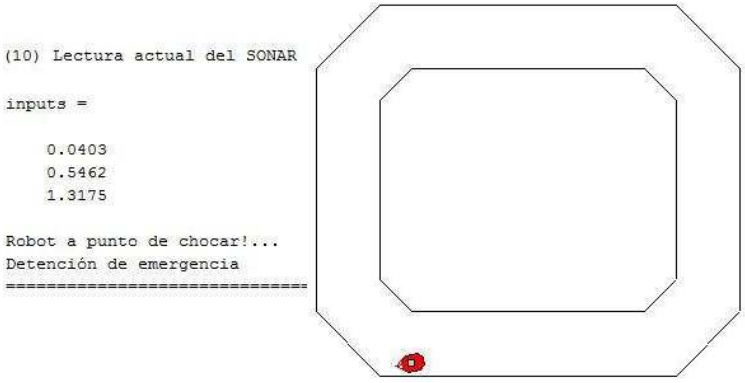


Figura 69. Error, choque inminente

7.2. Resultados del diseño del neurocontrolador

Los criterios de diseño, tanto del neurocontrolador como del interfaz, han sido fundamentados en las referencias bibliográficas y en pruebas de rendimiento.

7.2.1. Elección del perceptrón a utilizar

'feedforwardnet' vs. 'fitnet' vs. 'newff'

En el Toolbox se disponen de tres tipos de redes alimentadas hacia delante, que según Beale, Hagan y Demuth, autores del manual del Toolbox R2015a, ofrecen el mismo rendimiento cuando se resuelve una aproximación de una función, un *'data-fiting'* (sección 3.2.3).

Para saber que red tiene mejor rendimiento en el entrenamiento de esta aplicación, con estructura, parámetros y patrones definidos se realizó una comparación de tres entrenamientos entre las tres redes.

Criterios constantes:

- Número de capas: 1 oculta – 1 de salida
- Número de neuronas en capa oculta: 36
- Funciones de transferencia: *'tansig'* / *'purelin'*
- Función de entrenamiento: *'trainlm'*
- Función de división de patrones: *'dividetrain'*
- Número de patrones de entrenamiento: 50

De acuerdo a los criterios de aproximación de Barron, se decidió que la estructura de la red sea de una sola capa oculta y una de salida, y el número de neuronas en la capa oculta (sección 2.7.1). Además se calculó el error aproximado según Barron (sección 6.2.2).

Considerando el postulado de Cybenko y nuevamente el criterio de Beale, Hagan y Demuth, se decidió que la combinación de las funciones de transferencia (activación) sea: en capa oculta la función tangente sigmoide y en salida la función lineal (sección 2.7.1 y 3.2.2).

El número de patrones utilizados en las pruebas de rendimiento es de 50. Al no ser una cantidad muy grande se decidió usar todos en el entrenamiento, evitando hacer validación. De este modo se evita cortes del entrenamiento por controles de validación *'check Validation'* (sección 3.3.4).

Se decidió también usar el método de Levenberg-Marquardt por ser el más rápido en el entrenamiento de redes de pequeña y mediano tamaño, como la del proyecto actual (sección 3.3.3)

Resultados:

Tabla 26.

Comparación rendimiento 'feedforward'–'fitnet'-'newff'

Red	**Prueba**	**Rendimiento** (mse)	**Épocas**	**Parámetro** (corta entrenamiento)
'feedforwardnet'	1	1.53 e -16	151	Valor mínimo de gradiente *min_grad* encontrado
	2	2.17 e -13	353	Valor mínimo de gradiente *min_grad* encontrado
	3	4.78 e- 19	255	Valor mínimo de gradiente *min_grad* encontrado
'fitnet'	1	1.10 6 -7	229	Valor mínimo de gradiente *min_grad* encontrado
	2	9.66 e -13	205	Valor mínimo de gradiente *min_grad encontrado*
	3	2.24 e -8	170	Valor mínimo de gradiente *min_grad* encontrado
'newff'	1	1.13 e -15	379	Valor mínimo de gradiente *min_grad* encontrado
	2	2.37 e -7	356	Valor mínimo de gradiente *min_grad* encontrado
	3	2.41 e -13	244	Valor mínimo de gradiente *min_grad* encontrado

El rendimiento *'performance'* corresponde al valor mínimo del '*error cuadrático medio mse'* calculado en el entrenamiento de la red. Mientras menor es el *mse,* mejor es el entrenamiento (sección 3.3.4).

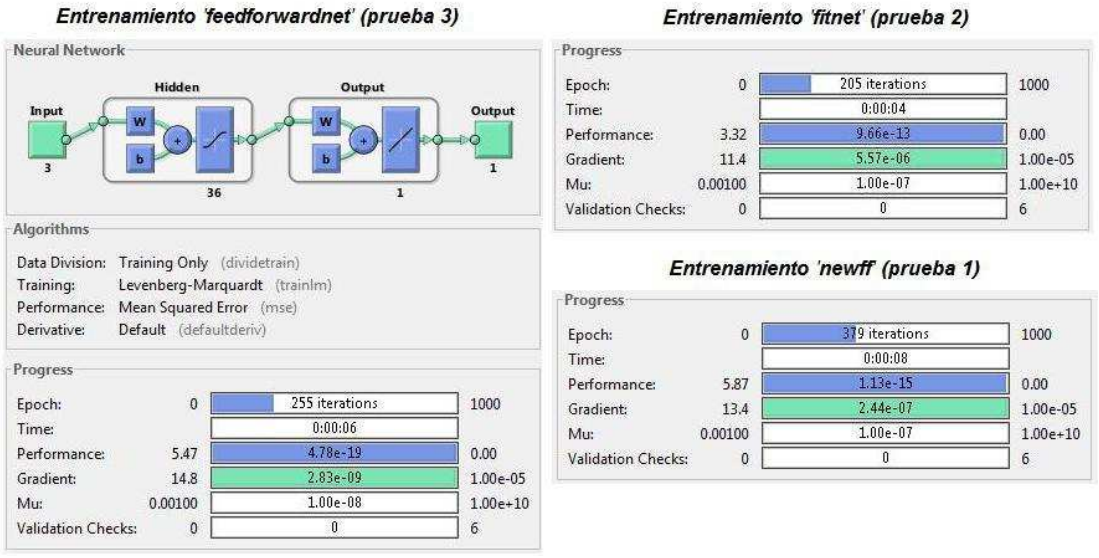


Figura 70. Parám. entrenamiento *'feedforwardnet'-'fitnet'–'newff'*

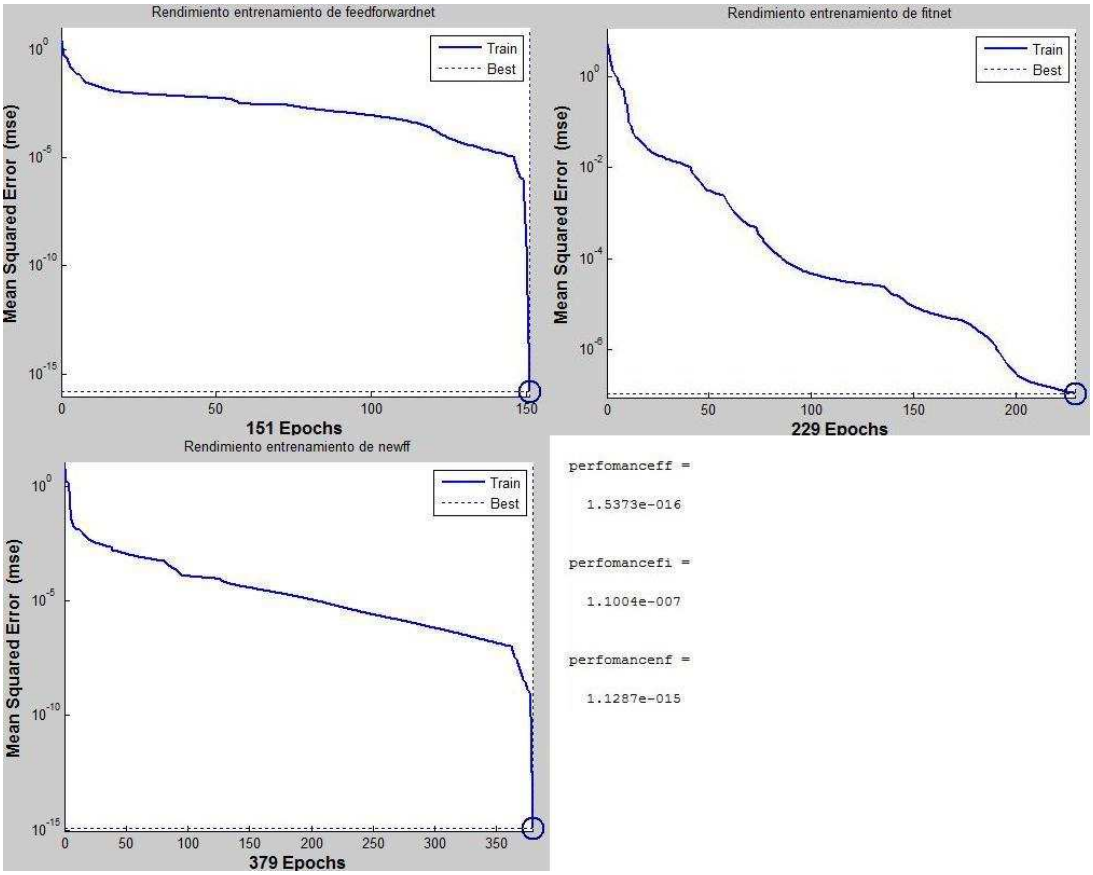


Figura 71. Gráficas rendimiento redes distintas - Prueba1

Los resultados muestran un leve mejor rendimiento de *'feedforwardnet'* sobre *'fitnet'*. La diferencia entre las dos puede ser una consecuencia de la toma de pesos iniciales aleatorios, diferentes en cada entrenamiento.

La *'fitnet'* es una versión especializada de *'feedforwardnet'* para aproximaciones, y la *'newff'* es la antecesora de *'feedforwardnet',* que a partir del Matlab R2010b es considerada obsoleta (sección 3.2.3).

En la Figura 70 y Figura 71 se muestran algunos resultados de las comparaciones hechas. Estos han sido resumidos todos en la Tabla 26.

7.2.2. Elección del número de neuronas ocultas

Número de Neuronas: 8 vs. 36 vs. 108

El *error cuadrático medio mse* calculado según el criterio de Barron, no deja de ser una aproximación (sección 6.2.2).

Por lo tanto, se decidió comparar el rendimiento de tres redes, una con 8 neuronas en la capa oculta (valor menor, al precargado en la red), una con 36 (valor calculado según criterio de Barron) y otra con 108 (valor tres veces mayor al calculado con el criterio).

Criterios Constantes:

- Tipo de red: *'feedforwardnet'*
- Número de capas: 1 oculta – 1 de salida
- Funciones de transferencia: *'tansig'* / *'purelin'*
- Función de entrenamiento: *'trainlm'*
- Función de división de patrones: *'dividetrain'*
- Número de patrones de entrenamiento: 50

Resultados:

Los resultados muestran que pocas neuronas (8) en la capa oculta merman el rendimiento de la red. El entrenamiento no estuvo nada cerca de converger y fue cortado a pesar de que e*l error cuadrático medio* fue alto, pues la cantidad máxima de iteraciones fue sobrepasada.

En el entrenamiento para esta aplicación, el rendimiento de una red de 108 neuronas ocultas es moderadamente mejor al de una de 36. Además emplea menos épocas, aunque estas son de más duración.

Un exceso de neuronas en la capa oculta produce un sobre-entrenamiento de la red (sección 3.4.3), y un mayor número de cálculos en cada iteración, lo que a su vez hace lento el entrenamiento. Una red sobre entrenada deja poco espacio a la generalización (Bullinaria, 2014).

Tabla 27.

Comparación de rendimiento 8-36-108 neuronas ocultas

Num.	**Prueba**	**Rendimiento** (mse)	**Épocas**	**Parámetro** (que corta el entrenamiento)
8 ocultas	1	0.0081	>1000	exceso de épocas
	2	0.0029	>1000	exceso de épocas
	3	9.01 e-4	>1000	exceso de épocas
36 ocultas	1	4.19 e -13	269	Valor mínimo de gradiente *min_grad* encontrado
	2	4.56 e -16	187	Valor mínimo de gradiente *min_grad encontrado*
	3	2.05 e-8	360	Valor mínimo de gradiente *min_grad* encontrado
108 ocultas	1	7.92 e-14	144	Valor mínimo de gradiente *min_grad* encontrado * Cada época se demora más
	2	5.17 e-13	39	Valor mínimo de gradiente *min_grad* encontrado * Cada época se demora más
	3	6.45 e-14	55	Valor mínimo de gradiente *min_grad* encontrado * Cada época se demora más

En la Figura 72 y en la Figura 73 se muestran algunos resultados de las

comparaciones hechas. Estos han sido resumidos todos en la Tabla 27.

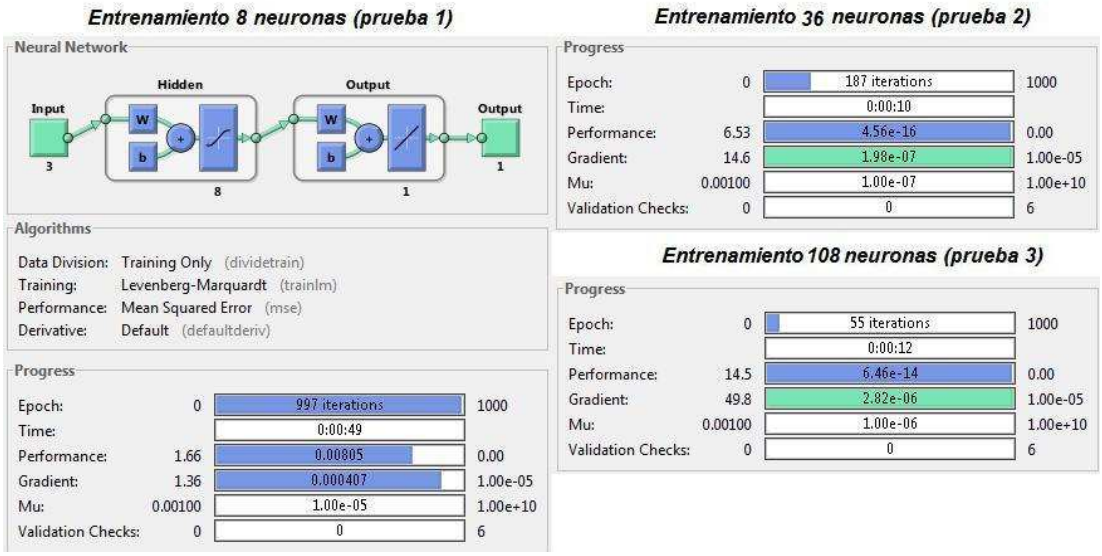


Figura 72. Parám. entrenamiento 8-36-108 neuronas ocultas

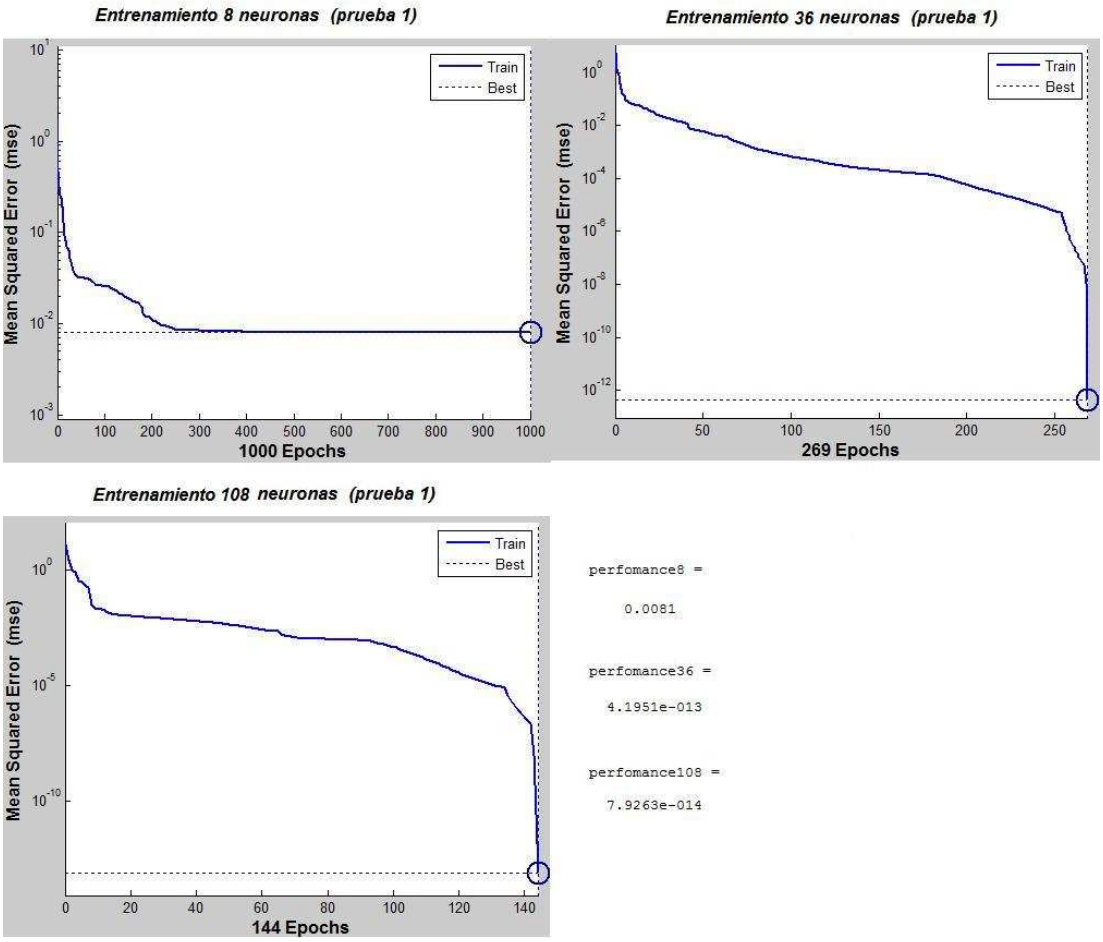


Figura 73. Gráficas de rendimiento neuronas ocultas - Prueba1

7.2.3. Elección del método de entrenamiento

'trainlm' – 'trainbfg' – 'trainoss'

Se evaluaron tres métodos rápidos: Levenberg-Marquardt *'trainlm'*, y dos Quasi-Newton *'trainbfg'* y *'trainoss',* para saber cuál de estos ofrece mejor rendimiento en el entrenamiento del neurocontrolador.

Criterios Constantes:

Excluyendo al número de patrones de entrenamiento, todos los demás elementos, evaluados en las comparaciones anteriores, fueron elegidos para el diseño definitivo. Los resultados, que se obtengan en ésta comparación, por lo tanto reflejan el rendimiento del entrenamiento del controlador final.

- Tipo de red: *'feedforwardnet'*
- Número de capas: 1 oculta – 1 de salida
- Número de neuronas en capa oculta: 36
- Funciones de transferencia: *'tansig'* / *'purelin'*
- Función de división de patrones: *'dividetrain'*
- Número de patrones de entrenamiento: 50

Resultados:

Según Beale, Hagan y Demuth, *'trainlm'* es más rápido y es altamente recomendado como primera opción para entrenamiento supervisado en redes de tamaño pequeño y mediano.

Esta premisa fue comprobada con los resultados obtenidos. De entre los tres, *'trainlm'* no solo es el algoritmo más rápido sino también el que ofrece, en promedio, un mejor rendimiento. *'trainbfg'* también ofrece un *error cuadrático mse* bajo*,* sin embargo necesita de mucho cálculo y de más iteraciones para converger.

Por otro lado, *'trainoss'* es el que ofrece los peores resultados entre los tres. El *error cuadrático medio mse,* que calcula, es demasiado alto. Lo que no garantiza una buena aproximación de la función. Además, entre los tres algoritmos es el que más se demora y más iteraciones necesita.

Tabla 28.

Rendimientos algoritmos 'trainlm'-'trainbfg'-'trainoss'

Num.	Prueba	Rendimiento (mse)	Épocas	Parámetro (que corta el entrenamiento)
'trainlm'	1	6.65 e-9	186	Valor mínimo de gradiente *min_grad* encontrado
	2	1.20 e-9	431	Valor mínimo de gradiente *min_grad* encontrado
	3	2.87 e-15	221	Valor mínimo de gradiente *min_grad* encontrado
'trainbfg'	1	1.34 e-4	>1000	exceso de épocas
	2	3.44 e-7	>1000	exceso de épocas
	3	5.89 e-14	755	Valor mínimo de gradiente *min_grad* encontrado
'trainoss'	1	0.0087	>1000	exceso de épocas
	2	0.0096	>1000	exceso de épocas
	3	0.0072	>1000	exceso de épocas

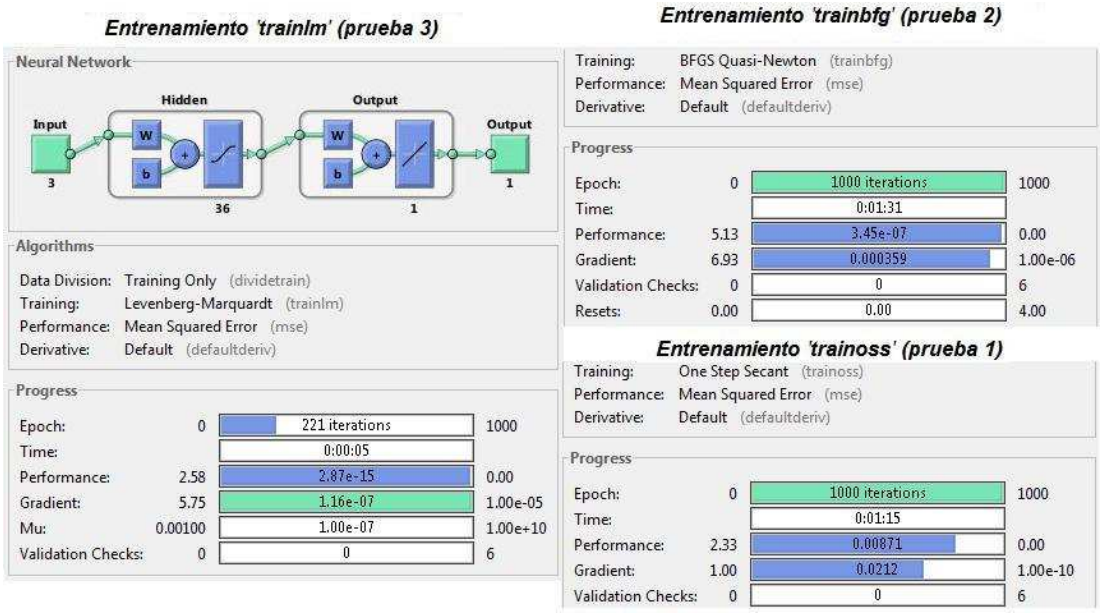


Figura 74. Parám. entrenamiento de algoritmos

En la Figura 74 y en la Figura 75 se muestran algunos resultados de las

comparaciones hechas. Estos han sido resumidos todos en la Tabla 28.

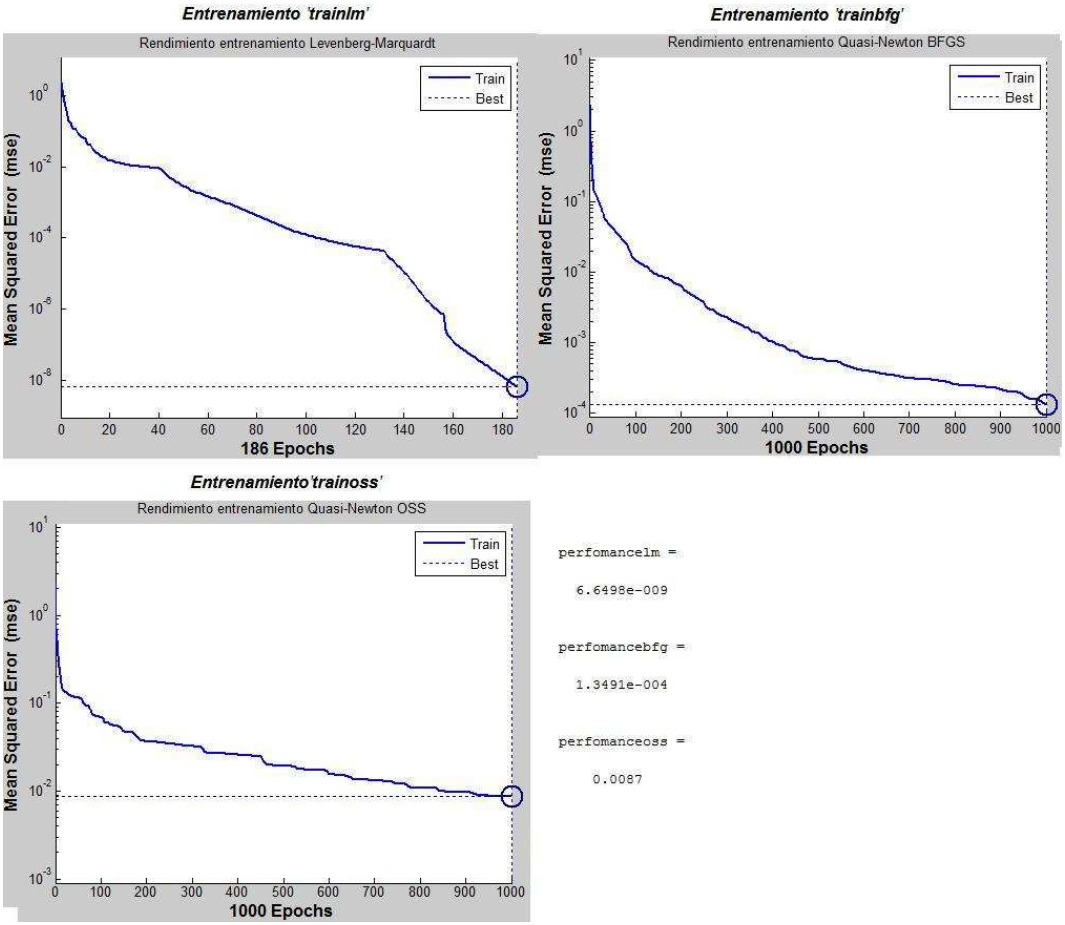


Figura 75. Gráficas de rendimiento algoritmos - Prueba1

Cabe recalcar que en ninguna de las evaluaciones hechas se ha conseguido la convergencia del entrenamiento. El *error cuadrático medio mse* nunca llego a ser cero, pero su valor con *'trainlm'* si llego a minimizarse, en un orden bajo, en la mayoría de comparaciones.

El algoritmo de entrenamiento rápido 'trainlm' está basado en una técnica de optimización numérica (sección 3.3.3). En general las técnicas de optimización numérica buscan minimizar el *mse* (sección 2.5).

Por lo tanto, los resultados obtenidos con *'trainlm'* fueron validados, no solo por el mínimo *mse* que logran sino también porque el valor de la gradiente en la región de cálculo es cercano a cero, lo que significa que un punto estacionario mínimo ha sido encontrado (sección 2.8.2).

Observación:

El parámetro, que ha cortado a la mayoría de entrenamientos, ha sido el valor mínimo de gradiente *min_grad*. Con la finalidad de conseguir un mejor rendimiento (*mse*) en el entrenamiento, se ha bajado este límite: de 1 e-05 (valor mínimo en *'trainlm'*) a 1 e-08.

```
netp3dx.trainParam.min_grad = 1e-08;
[netp3dx,tr]=train(netp3dx,inputs,targets);
```

Y en efecto el entrenamiento, con el nuevo *min_grad*, logra minimizar mucho mejor el error cuadrático medio *mse*.

7.3. Resultado de transferencia de datos MEX

La transferencia de datos fue desarrollada usando un archivo MEX, programado en Matlab. Dicho archivo fue exitosamente usado como puerta de enlace a la librería Aria. Se logró enlazar correctamente las funciones de las clases de Aria, las funciones de comandos de movimiento *'motion command functions'* y los comandos directos *'direct commands'* (sección 5.5.3).

7.3.1. Frecuencia del ciclo de control

La frecuencia del ciclo de control (Figura 66) ocasionó inconvenientes en el rendimiento de la interfaz de transferencia. Para evitar ciclos inconclusos y sobrepuestos, se calculó el intervalo aproximado de tiempo necesario para completar un ciclo.

Se han definido dos ciclos de acuerdo a los movimientos del robot, un ciclo para el movimiento 2 hacia delante en línea recta, y otro para los movimientos 1- 3 laterales.

Velocidad traslacional

Esta velocidad rige en el movimiento 2. La velocidad máxima (sección 1.7.3) no se logra en esta tarea porque los desplazamientos son cortos. Por lo que, se calculó una velocidad promedio con capturas realizadas del robot en movimiento.

```
>> frente()  >> frente()  >> frente()
>> getVel()  >> getVel()  >> getVel()

ans =        ans =        ans =

   210          181          180

>> getVel()  >> getVel()  >> getVel()

ans =        ans =        ans =

   529          452          432

>> getVel()  >> getVel()  >> getVel()

ans =        ans =        ans =

   290          302          372

>> getVel()  >> getVel()  >> getVel()

ans =        ans =        ans =

     0           29          192
```

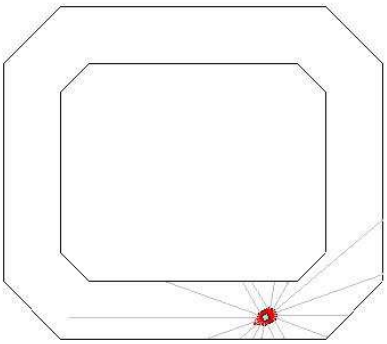


Figura 76. Capturas de velocidad traslacional

$$velPromedio = 264.33\ [mm/seg]$$

Velocidad rotacional

```
>> izq()        >> izq()        >> izq()
>> getRotVel()  >> getRotVel()  >> getRotVel()

ans =           ans =           ans =

    27              24              24

>> getRotVel()  >> getRotVel()  >> getRotVel()

ans =           ans =           ans =

    12              13              12

>> der()        >> der()        >> der()
>> getRotVel()  >> getRotVel()  >> getRotVel()

ans =           ans =           ans =

   -27             -24             -27

>> getRotVel()  >> getRotVel()  >> getRotVel()

ans =           ans =           ans =

   -10             -13             -13
```

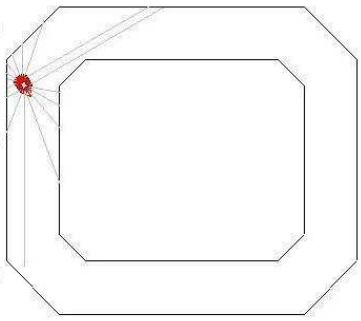


Figura 77. Capturas de velocidad rotacional

$$velRotPromedio = 18.83\ [grad/seg]$$

Como en la velocidad traslacional, en la rotacional se calculó también una velocidad promedio.

Tabla 29.

Duración aprox. ciclo - movimiento 2

Actividad de ciclo	Duración de actividad
Tiempo máximo de espera del paquete SIP del firmware 0.1 [seg] 3 SIPs esperados	t= 0.3 [seg]
Tiempo necesario para movimiento 2 distancia=350 [mm] velPromedio=264.33[mm/seg]	t= 1.33 [seg]
Tiempo aproximado de envió de paquetes de comando	t= 0.1 [seg]
Tiempo aproximado de un ciclo completo de movimiento 2	T= 1.73 [seg]

* Referencia paquete SPI y paquete de comando: sección 3.13

Tabla 30.

Duración aprox. ciclo - movimiento 1-3

Actividad de ciclo	Duración de actividad
Tiempo máximo de espera del paquete SIP del firmware 0.1 [seg] 3 SIPs esperados	t=0.3 [seg]
Tiempo necesario para movimiento 1 – 3 distancia=30 [grad] velPromedio=18.83[grad/seg]	t=1.59 [seg]
Tiempo aproximado de envió de paquetes de comando	t=0.1 [seg]
Tiempo aproximado de un ciclo completo movimientos 1 y 3	T= 1.99 [seg]

* Referencia paquete SPI y paquete de comando: sección 3.13

Considerando estas duraciones aproximadas, se procedió a otorgar dos segundos para cada ciclo de control. A pesar de que este tiempo cubre a los dos tipos de movimiento, la taza de pérdida de datos fue elevada, como lo muestra la Tabla 31.

Después de seguir un procedimiento empírico, se decidió darle un segundo más a cada ciclo, es decir en total tres segundos. Consiguiendo reducir marcadamente la taza de perdidas, y también prolongando la cantidad de ciclos correctos por ejecución.

Tabla 31.

Perdida de datos en ciclos de control

Duración de ciclo	Taza de perdidas	Características
2 [seg]	90%	En 9 de cada 10 ejecuciones del algoritmo de control se pierde datos de sensoramiento, especialmente después del decimo ciclo. (Ejemplo Figura 78)
3 [seg]	<10%	En 1 de cada 10 ejecuciones del algoritmo se pierde datos de sensoramiento, mayormente después del ciclo número 35.

En la Figura 78 se muestra un error: producto de una pérdida de datos en transferencia por un ciclo de control cortado. La lectura frontal de los sensores ultrasónicos no fue retornada, solo se cargo su rango máximo.

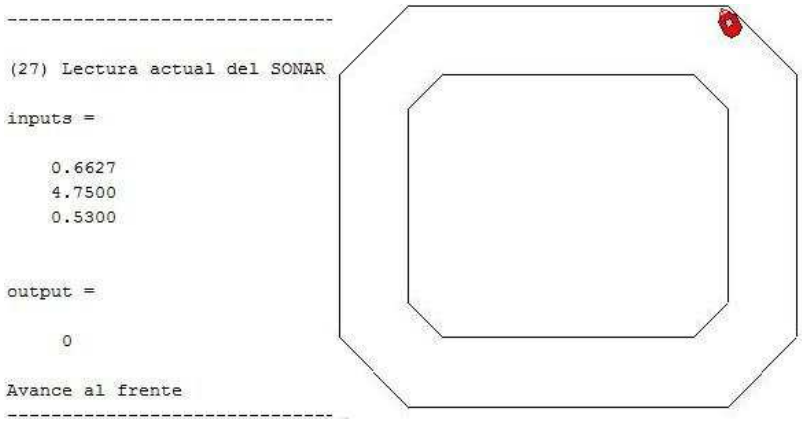


Figura 78. Error - pérdida de datos en transferencia

Todos los cálculos realizados fueron hechos con conexión del simulador MobileSim al puerto TCP. Si se tuviese que controlar al robot físico, se debería re calcular las duraciones, considerando otras variables propias de la conexión física y sus respectivas latencias: como las velocidades de transferencia serial o Ethernet.

El tiempo de respuesta del robot y el del controlador son considerados en cada ciclo de control. En el caso del robot, se debe considerar un tiempo de respuesta para las funciones de sensoramiento (mínimo de 100 milisegundos, que es el tiempo que necesita el microcontrolador para enviar el SIP con la información requerida), y uno de acción detallado en la segunda fila de las tablas 29 y 30. El tiempo de respuesta del

neurocontrolador está en función del robot, puesto que se necesita los datos de entrada, de sensoramiento, para avanzar en el ciclo de control.

Observaciones extras:

La función comando de movimiento *'move(distance)'* de la clase *ArRobot* tiende a mover al robot, hacia delante o hacia atrás, una distancia distinta a la declarada en el envió, en la mayoría de casos menor. En los movimientos laterales se invoca separadamente primero la función *'setDeltaHeading(+/-30°)'* y luego *'move(200)'*, sin embargo este par de movimientos se sobreponen en la actuación como si se tratase de un giro abierto.

7.4. Resultado Global Comportamiento del Robot

La recolección de patrones de entrenamiento juega un papel fundamental en el rendimiento global del neurocontrolador. Con el objetivo de tener suficientes patrones, que sean además ejemplos idóneos para un comportamiento autónomo e inteligente del robot, se recolecto un primer juego de 25, luego uno de 50 y finalmente uno de 90 (o más patrones).

Tabla 32.

Resultados globales – control de trayectoria

Num. Patrones	Prueba	Ciclos ejecutados	Características
25	1	5	El robot inicia aproximando bien, pero en muy pocos ciclos. Se presentan fallas ya sea por una mala aproximación o también porque las entradas actuales no fueron consideradas en el rango de entrenamiento.
	2	11	
	3	8	
50	1	22	El robot muestra un comportamiento más robusto ante distintas entradas, aunque sigue siendo débil. En algunos casos se presentan fallas por mala aproximación, pero las originadas por entradas no consideradas en entrenamiento se presentan más a menudo.
	2	18	
	3	25	
+90	1	56	El robot consigue darse una vuelta completa en la pista 1, en ambas direcciones. Las fallas por mala aproximación se minimizan, no obstante las de entradas fuera de rango no disminuyen en la misma magnitud.
	2	45	
	3	51	

El neurocontrolador fue entrenado inicialmente en la pista 1, hasta que el robot consiguió dar una vuelta completa, en ambas direcciones y con posiciones iniciales distintas. Después se probó el entrenamiento en la pista 2 obteniendo resultados iniciales deficientes.

Tras una recolección de pocos patrones en la segunda pista, el rendimiento del neurocontrolador subió. No obstante, para determinadas regiones de la pista 2 el entrenamiento sigue siendo poco aceptable.

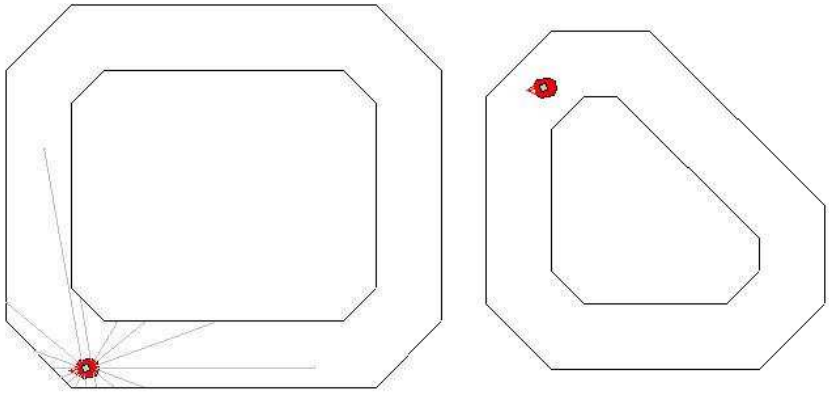


Figura 79. Pista 1 - Pista 2

7.5. Discusión

Los resultados obtenidos en este proyecto muestran que:

Un perceptrón multicapa, creado, entrenado y simulado en el Neural Network Toolbox de Matlab, fue usado exitosamente como neurocontrolador de la plataforma robótica móvil Pioneer P3-DX en el problema de control de trayectoria.

Este problema, que fue planteado como una aproximación de una función real no lineal, fue resuelto con una red alimentada hacia delante *'feedforwardnet' entrenada* mediante el algoritmo de optimización de Levenberg-Marquardt *'trainlm'*.

El entrenamiento no convergió, pues el error cuadrático medio nunca

llegó a ser cero, sin embargo; después de revisar la definición de optimización numérica de la IMFORMS Computing Society (es el proceso sistemático de seleccionar el mejor elemento de un conjunto que maximice o minimice una determinada función real) se validó el entrenamiento, pues los pesos seleccionados han minimizado en un orden muy bajo el error cuadrático medio calculado. Además, el valor del gradiente es lo suficientemente bajo en la región de cálculo, lo que significa que un punto estacionario mínimo fue encontrado.

El algoritmo de Levenberg-Marquardt *'trainlm'* y los algoritmos Quasi-Newton *'trainbfg'* - *'trainoss'* son entrenamientos rápidos de retropropagación del error, que aproximan el cálculo de la matriz de gradientes de segundo orden *Hessiana,* evitándolo en las redes *'feedforwardnet'* porque se torna complejo y lento, según Demuth, Beale y Hagan, autores de la guía de usuario del Neural Network Toolbox. Por lo tanto, el perceptrón multicapa no fue entrenado, puntualmente, por un método de descenso de gradiente de segundo orden, sino por uno de entrenamiento rápido con aproximación de gradiente de segundo orden.

De entre los tres, conforme a los resultados mostrados en la Tabla 28, el método de Levenberg-Marquardt es el que mejores resultados entregó, porque fue el más rápido y el de mejor rendimiento, es decir el que calculó el error cuadrático medio más bajo. Confirmando los resultados del análisis presentado en el capítulo 8 'Selección de una función de entrenamiento para una red neuronal multicapa' del Toolbox.

El control de trayectoria se corrió directamente desde la ventana de comandos de Matlab R2014a. Este proyecto es una guía de práctica enfocada en el análisis de los algoritmos de desarrollo del neurocontrolador y del interfaz de transferencia de datos Matlab – Aria. Se evitó el uso de interfaces de usuario, precisamente para agilizar su comprensión y recreaciones posteriores.

Discusión del esquema de diseño del neurocontrolador

El neurocontrolador fue diseñado bajo un esquema de control directo, aproximando un modelo de control adaptativo directo, basado en un modelo de referencia del entorno. Las entradas de la red son directamente recibidas desde la plataforma, no obstante la salida de la red es de referencia (Tabla 23) aunque en el mismo algoritmo de control es transformado en señales de control. Esta característica del neurocontrolador genera un espacio, para analizar si desde algún enfoque se debe redefinir el esquema de control.

Discusión de la dinámica de control

El control del robot se realizó en dinámica lenta con una frecuencia: de un ciclo de control por cada tres segundos, realizando el sensoramiento en estado estacionario de los valores de la posición del robot respecto de las paredes, sin considerar ningún transitorio durante cada ciclo de control. Una de las principales razones, que motivaron la elección de un esquema de control inteligente, fue el desconocimiento de la dinámica de la plataforma robótica y de su respuesta ante el problema planteado. Sin embargo, una vez descubiertas las imprecisiones de las acciones de control propuestas (sección 7.3 observaciones extras), estas justifican una dinámica de control más compleja.

La frecuencia de control del robot 0.5 [Hz], matemáticamente calculada para la simulación fue demasiada alta, presentándose ciclos de control sobrepuestos y cortados, por lo que se debió reducirla a 0.33 [Hz]. En los cálculos matemáticos solo se consideró el tiempo de respuesta del microcontrolador de la plataforma, más no las posibles latencias del sistema operativo en el que se ejecuta el neurocontrolador y el interfaz, así como las posibles latencias que se generen en el robot.

Discusión de recolección de patrones de entrenamiento

El entrenamiento, con los atributos especificados en la sección 7.2.3 –

criterios constantes, ha ofrecido resultados positivos. Sin embargo, el rendimiento del neurocontrolador es bueno pero no tanto como se esperaba. Considerando la advertencia que realizan Beale, Demuth y Hagan y Kriesel (sección 6.2.1), se puede afirmar que la recolección de patrones de entrenamiento fue errada, porque estos no reflejan los escenarios correctos de ejemplo, ni tampoco son suficientes. Las entradas usadas en el entrenamiento no cubrieron todo el rango de posibles entradas, y como la red no puede extrapolar las aproximaciones erradas se produjeron en gran cantidad. Además, como los patrones no son suficientes, la red no pudo generalizar correctamente un comportamiento inteligente.

7.6. Trabajos futuros

Después de la discusión se puede decir que se han presentado dos dificultades, que merecen una solución en trabajos futuros a partir de los resultados del proyecto actual:

- Una definición más precisa de la dinámica de control: estudiar todos los retardos que generan la latencia total en el sistema de control, considerando tanto los de la plataforma robótica, del microcontrolador y el firmware, de los actuadores: motores y de los sensores: arreglo sonar, como los del sistema operativo donde se corre el neurocontrolador, y los de la comunicación, para plantear una dinámica de control más completa. Estudiar también si los retardos son mayores cuando se trabaja con el robot físico. Comprobar si se puede obtener un sistema de control más eficiente, desarrollando el neurocontrolador en otro sistema operativo o embebiéndolo en algún dispositivo, como una tarjeta FPGA ‘field programmable gate array’.

- Otra recolección de patrones de entrenamiento: analizar una técnica que permita definir la cantidad de patrones necesarios para mejorar el rendimiento del neurocontrolador, sin que estos sean pocos y se produzca un entrenamiento pobre, y sin que sean demasiados y se

llegue a un sobre entrenamiento. Si se demuestra que el neurocontrolador definitivamente es incapaz de ofrecer mejores resultados para el problema propuesto, estudiar otros esquemas ya sean de control inteligente, como un controlador neuro-difuso o una red inmune idiotípica, o de control moderno, como un filtro de Kalman, aunque se debe prestar especial atención a la función no lineal analizada.

8. CONCLUSIONES Y RECOMENDACIONES

8.1. Conclusiones

1. El perceptrón multicapa creado, entrenado y simulado con el Neural Network Toolbox del Matlab R2014a puede controlar la trayectoria con la que se mueve la plataforma robótica móvil Pioneer P3-DX dentro de una pista simulada en MobileSim de 1.5 metros de ancho con tramos rectos y curvas de 45 grados.

2. El perceptrón multicapa alimentado hacia delante con una capa oculta de 36 neuronas y un capa final de una neurona, tiene la arquitectura capaz de conseguir un error cuadrático medio en el orden de las millonésimas durante el entrenamiento, sin que el perceptrón se sobre entrene; considerando el criterio de aproximación de Andrew Barron y los resultados registrados en la Tabla 27, de las pruebas de rendimiento realizadas.

3. La combinación de funciones de activación o transferencia, tangente sigmoide en la capa oculta y lineal en la capa de salida, permite obtener el mejor resultado cuando se resuelve un problema de aproximación de una función con un perceptrón, porque la función tangente sigmoide al ser no lineal aprende la relación no lineal entre los vectores de entrada y salida, y al ser diferenciable facilita el cálculo del gradiente de primer orden y de segundo orden en el entrenamiento; de acuerdo a las indicaciones publicadas en la guía de usuario del Neural Network Toolbox del Matlab R2014a.

4. El perceptrón es entrenado en aprendizaje supervisado y por retropropagación del error, con más de 90 patrones de entrenamiento, en un proceso sin validación, a través del método de Levenberg-Marquardt 'trainlm', que es el más rápido y el que menor error cuadrático medio calcula entre los tres métodos que aproximan el cálculo del gradiente de segundo orden, como se muestra en la Tabla 28.

5. El controlador se diseñó aproximando un sistema de control de tipo adaptativo directo: basado en un modelo de referencia del entorno, simulado en MobileSim 0.7.2-1.

6. El archivo MEX binario es compilado desde un archivo MEX código fuente en Matlab R2014a con el compilador C++ de Microsoft Visual C++ 2010 Service Pack 1, en un ordenador de arquitectura de 32 bits con sistema operativo Microsoft Windows 7 Service Pack 1.

7. El archivo MEX binario permite la transferencia de datos entre el controlador neuronal desarrollado en Matlab R2014a y el firmware ARCOS del microcontrolador de la plataforma robótica, enlazándoles a través de las funciones de la librería ARIA 2.7.6, precompiladas en C++.

8.2. Recomendaciones

1. Se recomienda usar el sistema operativo y los programas computacionales indicados en la Tabla 11, pues la compatibilidad entre los compiladores de Microsoft Visual C++ 2010 y Matlab R2014a dio excelentes resultados.

2. La arquitectura del ordenador empleado en el desarrollo y la ejecución de la aplicación es de 32 bits, por lo que se sugiere elegir adecuadamente el software; si el ordenador a disposición es de 64 bits.

3. Se advierte que la recolección de patrones no cubrió todo el rango

posible de entradas para el entrenamiento, y como la red no puede extrapolar respuestas, en promedio, una de cada diez salidas ha sido generalizada incorrectamente; por lo que se recomienda revisar la lógica de recolección y los patrones seleccionados que están a disposición en la carpeta “extras” adjunta a este documento.

4. Se recomienda el uso de este documento como texto auxiliar en el aprendizaje de la teoría de Control Neuronal, y también como guía de práctica, en la asignatura de Control Inteligente, de la Carrera de Ingeniería en Electrónica, Automatización y Control de la ESPE, pues ha sido íntegramente referenciado en las fuentes bibliográficas enlistadas, y detalladamente redactado, registrando cada uno de los pasos del desarrollo de la aplicación.

REFERENCIAS BIBLIOGRAFICAS

- Adept MobileRobots. (31 de mayo de 2012). *ARIA Developer's Reference Manual*. Recuperado el 03 de noviembre de 2014, de http://robots.mobilerobots.com/Aria/docs/main.html
- Adept MobileRobots. (2011). *Pioneer P3-DX Overview*. Recuperado el 22 de febrero de 2015, de Research Robots: http://www.mobilerobots.com/researchRobots/PioneerP3DX.aspx
- Axelsson, M. (22 de Octubre de 2007). *An Introduction to MATLAB MEX-files.* Recuperado el 14 de abril de 2015, de http://pages.cs.wisc.edu/~ferris/cs733/Seminarium_CBA_ht07_MATLAB_MEX_handouts.pdf
- Babuska, R. (2001). *Fuzzy and Neural Control.* Delft: Delft University of Technology.
- Beale, M. H., Hagan, M. H., & Demuth, H. B. (marzo de 2015). *User's Guide Neural Network Toolbox MATLAB R2015a.* Recuperado el 20 de mayo de 2015, de MathWorks: https://www.mathworks.com/help/pdf_doc/nnet/nnet_ug.pdf
- Beale, M. H., Hagan, M. T., & Demuth, H. B. (marzo de 2015). *Reference Neural Network Toolbox MATLAB R2015a.* Recuperado el 20 de mayo de 2015, de MathWorks: https://www.mathworks.com/help/pdf_doc/nnet/nnet_ref.pdf
- Beale, M., Hagan, M., & Demuth, H. (marzo de 2015). *Getting Started Guide Neural Network Toolbox MATLAB R2015a.* Recuperado el 20 de mayo de 2015, de MathWorks: https://www.mathworks.com/help/pdf_doc/nnet/nnet_gs.pdf
- Borgström, J. (3 de Abril de 2005). *ARIA and Matlab Integration with Applications.* Recuperado el 10 de abril de 2015, de Umeä University: https://www8.cs.umu.se/education/examina/Rapporter/JonasBorgstrom.pdf
- Bullinaria, J. A. (2014). *Introduction to Neural Computation, Neural*

Computation. Recuperado el 15 de Marzo de 2015, de http://www.cs.bham.ac.uk/~jxb/inc.html

- Ceballos Sierra, J. (1995). *Curso de Programación c / C++.* Madrid: RA-MA Editorial.
- Cengiz, Ö. (2003). *Classical Models of Neural Networks.* Recuperado el 28 de abril de 2015, de Neural Networks and Pattern Recognition: https://www.byclb.com/TR/Tutorials/neural_networks/ch8_1.htm
- Cheok, K. C. (12 de Diciembre de 2002). *Intelligent Control Systems Methods.* Recuperado el 05 de abril de 2015, de Oakland University: http://groups.engin.umd.umich.edu/vi/w2_workshops/Intel%20Cntrl%20Sys%20_cheok_w2.pdf
- Cotero Ochoa, J. B. (2005). *Control Clásico y Control Inteligente.* Recuperado el 05 de abril de 2015, de http://noticias.espe.edu.ec/daguffante/files/2012/10/Descargar.pdf
- Cybenko, G. (1989). *Mathematics of Control, Signals, and Systems - Aproximation by Superpositions of a Sigmoidal Functions.* New York: Springer.
- Demuth, H., & Beale, M. (1998). *Neural Network Toolbox-User's Guide* (3th ed.). Natick: The MathWorks, Inc.
- Fausett, L. (1994). *Fundaments of Neural Networks: Architectures, Algorithms and Applications.* Prentice Hall.
- Fernandez de Cañete, J. (s.f.). *Sistemas de Control Neuronal.* Recuperado el 06 de abril de 2015, de http://www.isa.uma.es/C8/Presentaciones%20de%20Clase/Document%20Library/Introduccion%20Sistemas%20de%20Control%20Neuronal.pdf
- Flores, M., & Proaño, A. (2013). *Evolución Artificial y Robótica Autónoma desarrollada en el Robot P3-DX con Aproximación basada en Comportamientos.* Sangolquí: ESPE.
- Fogel, D. B. (2006). *Evolutionary Computation, Toward a New Philosophy of Machine Intelligence.* (J. W. Inc., Ed.) Hoboken & New York: IEEE PRESS.
- Getreuer, P. (April de 2010). *Writing MATLAB C / MEX Code.*

Recuperado el 10 de abril de 2015, de https://classes.soe.ucsc.edu/ee264/Fall11/cmex.pdf

- Guffanti, D. (2013). *Control Remoto por Voz del Robot Pionner P3-DX.* Sangolquí: ESPE.
- Gurney, K. (2007). *An introduction to neural networks.* Londres: UCL Press.
- Hagan, M. T., Demuth, H. B., Beale, M. H., & De Jesús, O. (2012). *Neural Network Design.* Recuperado el 05 de mayo de 2015, de 2nd Edition: http://hagan.okstate.edu/NNDesign.pdf
- Haykin, S. (2005). *Neural Networks - A Comprehensive Foundation.* Patparganj: Pearson.
- IEEE-Chile. (4 de Noviembre de 2003). Newsletter of the IEEE Conputational Society. *IEEE Connections , 1* .
- Kriesel, D. (2007). *A Brief Introduction to Neural Networks.* Bonn: http://www.dkriesel.com/en/science/neural_networks.
- Kruse, R. (2009). *Neural Networks.* Recuperado el 28 de mayo de 2015, de Computational Intelligence Group - Faculty fot Computer Science - Universität Magdeburg: http://fuzzy.cs.uni-magdeburg.de/ci/nn/v09_svm_en.pdf
- Larson, R., & Edwards, B. (2010). *Calculus - Early Transcendental Functions* (Quinta ed.). Boston: Cencage Learning.
- Martínez Verdú, J. (2011). *Control Adaptativo.* Alicante: MITIT.
- MathWorks. (marzo de 2015). *C/C++, Fortran, and Phython API Reference - MATLAB R2015a.* Recuperado el 22 de mayo de 2015 , de http://www.mathworks.com/help/pdf_doc/matlab/apiref.pdf
- MathWorks. (marzo de 2015). *External Interfaces MATLAB R2015a.* Recuperado el 22 de mayo de 2015, de https://www.mathworks.com/help/pdf_doc/matlab/apiext.pdf
- MathWorks. (marzo de 2015). *Release Notes Neural Network Toolbox MATLAB R2015a.* Recuperado el 20 de mayo de 2015, de http://cn.mathworks.com/help/pdf_doc/nnet/rn.pdf
- MathWorks-Help. (marzo de 2015). *Advanced Software Development*. Recuperado el 02 de junio de 2015, de Documentation Matlab:

http://es.mathworks.com/help/matlab/software-development.html

- MathWorks-Support. (2015). *System Requirements and Supported Compilers*. Recuperado el 20 de marzo de 2015, de http://www.mathworks.com/support/sysreq/previous_releases.html
- Matich, D. J. (2001). *Redes Neuronales: Conceptos Básicos y Aplicaciones.* Rosario: Universidad Tecnológica Nacional - Facultad Regional Rosario.
- Microsoft-Support. (20 de julio de 2013). *Visual Studio .NET 2003 Readme (parte 1)*. Recuperado el 20 de marzo de 2015, de https://support.microsoft.com/es-es/kb/822354/es
- Microsoft-Support-Visual-Studio. (2010). *Requisitos de software para las características de Visual Studio*. Recuperado el 20 de marzo de 2015, de https://msdn.microsoft.com/es-es/library/vstudio/gg265786(v=vs.100).aspx#bkmk_vseditions
- MobileRobots - ActivMedia Robotics. (2006). *Operations Manual Pioneer 3.* Amherst: MobileRobots Inc.
- Morales, J., & Jaramillo, D. (2010). *Desarrollo de Aplicaciones y Documentación de las Plataformas Robóticas Pioneer P3-DX y Pioneer P3-AT.* Sangolquí: ESPE.
- National Science Foundation NFS: White, David A.; Sofge, Donald A. (1992). *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptative Approaches.* New York: Van Nostrand Reinhold.
- Özkan, C., & Sunar Erbek, F. (2003). The Comparison of Active Functions for Multispectral Landsat TM Image Classification. *Photogrammetric Enginnering & Remote Sensing Vol.69 No.11* , 1225-1234.
- Palacios Burgos, F. J. (2003). *Redes Neuronales con GNU/Linux.* Recuperado el 04 de abril de 2015, de Universidad Nacional de Salta - Argentina: http://softwarelibre.unsa.edu.ar/docs/descarga/2003/curso/htmls/redes_neuronales/c35.html
- Patnaik, S. (2007). *Robot Cognition and Navigation, An Experiment with Mobile Robots.* Berlín & Heidelberg: Springer - Verlag.

- Posada, L. (2009). *Matlab - Aria Interface*. Recuperado el 10 de abril de 2015, de http://www.rst.e-technik.tu-dortmund.de/cms/de/Forschung/Software/index.html
- Síma, J. (1998). *Introduction to Neural Network.* Praga: Instituto de Ciencia de la Computación, Academía de la Ciencia de la República Checa .
- Suttisinthong, N., & Seewirote, A. (2014). *Selection of Proper Activation Functions in Back-propagation Neural Network algorithm for Single-Circuit Transmission Line.* Hong Kong: IMEC.
- Universidad de Sevilla. (2012). *Conceptos básicos sobre Redes Neuronales.* Recuperado el 04 de abril de 2015, de Departamento de Matematica Aplicada I: http://grupo.us.es/gtocoma/pid/pid10/RedesNeuronales.htm
- Weisstein, E. (2015). *Wolfram MathWorld*. Recuperado el 20 de mayo de 2015, de Partial Derivative: http://mathworld.wolfram.com/
- Whitbrook, A. (2010). *Programming Mobile Robots with Aria and Player, A Guide to C++ Object - Oriented Control .* London: Springer - Verlag.
- Zurada, J. M. (1992). *Introduction to Artificial Neural Systems.* St. Paul: WEST PUBLISHING COMPANY.

Acta de Entrega

UNIVERSIDAD DE LAS FUERZAS ARMADAS - ESPE

INGENIERÍA EN ELECTRÓNICA, AUTOMATIZACIÓN Y CONTROL

ACTA DE ENTREGA

El proyecto titulado “Control Neuronal del Robot Móvil Pioneer P3–DX mediante un perceptrón multicapa implementado en Matlab” fue entregado al Departamento de Eléctrica y Electrónica, y reposa en la Universidad de las Fuerzas Armadas – ESPE, desde:

Sangolquí, julio del 2015

ELABORADO POR.

_____

Paúl Danilo Santillán Robalino

CI. 1712865250

AUTORIDAD

_____

Ing. Luis Orozco MSc.

DIRECTOR DE LA CARRERA DE INGENIERÍA EN ELECTRÓNICA AUTOMATIZACIÓN Y CONTROL