HERRAMIENTA DE APLICACIÓN SOTFWARE PARA CLASIFICACIÓN DE PATRONES DE DATOS IMPLEMENTANDO UNA ARQUITECTURA NEURO-FUZZY

FABIO ANDRES CABALLERO ESTEBAN
JULIAN SEBASTIAN SALAMANCA ESPINOSA

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE CIENCIAS FISICO-MECÁNICAS
ESCUELA DE INGENIERÍA MECÁNICA
BUCARAMANGA
2011

HERRAMIENTA DE APLICACIÓN SOFTWARE
PARA CLASIFICACIÓN DE PATRONES DE DATOS
IMPLEMENTANDO UNA ARQUITECTURA
NEURO-FUZZY

FABIO ANDRÉS CABALLERO ESTEBAN
JULIAN SEBASTIAN SALAMANCA ESPINOSA

Proyecto de grado para optar el título de Ingeniero Mecánico

Director
JORGE ENRIQUE MENESES FLOREZ
Ingeniero Mecánico

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE CIENCIAS FISICO-MECÁNICAS
ESCUELA DE INGENIERÍA MECÁNICA
BUCARAMANGA
2011

DEDICATORIA

A mis padres por su gran apoyo y confianza que me han brindado.
A mi padrino ELVER por los valores que me ha inculcado y por darme el apoyo de un padre .
A mis hermanos por su colaboración y dedicación que me ofrecieron.
A Erika por la motivación brindada.
A mis amigos y todas las personas con las que compartí en esta etapa de mi vida

FABIO ANDRÉS

DEDICATORIA

Este es un pequeño triunfo que quiero dedicar a mis padres por seguir adelante, a mis hermanos que a pesar de las diferencias siguen conmigo y en especial a mi hijo Sebastian por la motivación que me ha dado.

JULIAN SEBASTIAN

AGRADECIMIENTOS

Queremos expresar nuestros sinceros agradecimientos:

Al profesor Jorge Enrique Meneses por las bases académicas que nos brindó, y la guía que nos ofreció durante este proceso.

A nuestros padres Nubia, Luis y Joselin, Gladys por su apoyo incondicional y constante en este largo proceso.

A nuestros compañeros y amigos que aportaron sus ideas y consejos para que este proyecto saliera adelante en especial a Laura Herrera, Erika Gonzalez , Diana Tapias, Oscar Ebrath, Jefferson Villamizar, Diego Vera, Rafael Ramirez, Jose Jimenez.

Gracias a los campañeros del Laboratorio de Automatización Industrial por estar siempre en el momento indicado y por compartir gratos momentos.

A los profesores de la escuela de Ingeniería Mecánica de la Universidad Industrial de Santander por la formación profesional y social que nos dieron.

CONTENIDO

INTRODUCCION **19**

1 ALCANCES DEL PROYECTO **20**

1.1 IDENTIFICACIÓN DEL PROBLEMA 20

1.2 JUSTIFICACIÓN 21

1.3 OBJETIVOS 21

1.3.1 Objetivo general 21

1.3.2 Objetivos específicos 22

1.4 JUSTIFICACIÓN DE LA SOLUCIÓN 23

1.4.1 Alternativas de diseño 23

1.4.1.1 Arquitectura ANFIS 23

1.4.1.2 Arquitectura Perceptrón fuzzy (NEFCLASS) 24

1.4.1.3 Arquitectura Perceptrón fuzzy (NEFCON) 24

1.4.1.4 Arquitectura Perceptrón fuzzy (NEFPOX y NFIDENT) 24

1.4.2 Alternativas de desarrollo 25

1.4.2.1 Lenguaje de programación C 25

1.4.2.2 Lenguaje de programación M 25

1.4.2.3 Lenguaje de programación *JAVA* 26

1.4.2.4 Lenguaje de programación C++ 26

1.4.3 Alternativa seleccionada 26

2 AGENTES INTELIGENTES **28**

2.1 TÉCNICAS DE INTELIGENCIA ARTIFICIAL 30

2.1.1 Redes Neuronales Artificiales 30

2.1.2 Sistemas basados en Lógica Fuzzy. 31

2.1.3 Algoritmos genéticos 31

2.1.4 Sistemas Híbridos Inteligentes 32

2.2 SISTEMAS NEURO-FUZZY 32

2.2.1 Modelos Neuro-Fuzzy 33

2.2.1.1 Modelos cooperativos Neuro-Fuzzy 33

2.2.1.2 Modelos híbridos Neuro-Fuzzy. 34

3 MODELO NEFCLASS **37**
3.1 ARQUITECTURA DEL MODELO NEFCLASS 37
3.2 CONFIGURACIÓN DEL MODELO 40
3.3 PREPROCESAMIENTO 41
3.4 Entrenamiento 42

4 NEFCLASS-Q **43**
4.1 CREACIÓN DE LA RED NEURO-FUZZY 49
4.1.1 Creación de los conjuntos fuzzy (I) 49
4.1.2 Creación de Reglas(II) 50
4.1.3 Entrenamiento de la red(III) 54
4.1.4 Validación del entrenamiento (IV) 61
4.1.5 Clasificación (V) 62
4.2 DOCUMENTACIÓN DEL CÓDIGO 62
4.3 FUNCIONES DE NEFCLASS-Q 69
4.4 LIBRERÍA *FUZZYSET* 70
4.4.1 Constructores *FuzzySet* 70
4.4.1.1 Constructor por defecto 70
4.4.1.2 Constructores con parámetros 71
4.4.2 Métodos 74
4.4.3 Sobrecarga de operadores 82
4.4.3.1 Operador de flujo 82
4.4.3.2 Operador de comparación 83
4.4.3.3 Operador de asignación 83
4.5 LIBRERÍA *UNIVERSE.* 84
4.5.1 Constructores 84
4.5.1.1 Constructor por defecto 84
4.5.1.2 Constructor de copia 84
4.5.1.3 Destructor 85
4.5.2 Métodos 85
4.5.3 Sobrecarga de operadores 93
4.5.3.1 Operador de Flujo 93
4.5.3.2 Operador de asignación 94
4.6 LIBRERÍA *RULENEURON* 94
4.6.1 Constructores 95
4.6.1.1 Constructor por defecto 95
4.6.1.2 Constructor 95
4.6.1.3 Constructor 95
4.6.1.4 Constructor de copia 96

4.6.1.5 Destructor 97
4.6.2 Métodos 97
4.6.3 Funciones de obtención de datos 102
4.6.4 Sobrecarga de operadores 107
4.6.4.1 Operador de flujo 107
4.6.4.2 Operador de comparación 108
4.6.4.3 Operador de asignación 108
4.7 LIBRERÍA *CLASSNEURON.* 109
4.7.1 Constructor 109
4.7.1.1 Constructor por defecto 109
4.7.2 Métodos 109
4.8 LIBRERÍA *NETWORK* 111
4.8.1 Constructores 111
4.8.1.1 Constructor por defecto 111
4.8.1.2 Constructor 111
4.8.1.3 Constructor 112
4.8.1.4 Constructor 112
4.8.1.5 Constructor 112
4.8.1.6 Constructor 113
4.8.2 Métodos 113
4.9 CÓMO PROGRMAR CON LAS LIBRERÍAS? 132

5 CÓMO UTILIZAR NEFCLASS-Q? 135

6 CÓMO UTILIZAR LAS LIBRERIAS NEFCLASS-Q? 147

7 PRUEBA DE NEFCLASS-Q CON IRIS.DAT 152
7.1 PRUNING SIMPLE. 153
7.1.1 Resultados para el método pruning simple. 154
7.2 PRUNING DE LAS MEJORES REGLAS 154
7.2.1 Resultados con pruning de las 15 mejores reglas. 154
7.2.2 Resultados con pruning de las 10 mejores reglas. 155
7.3 PRUNING DE LAS MEJORE REGLAS POR CLASE 155
7.3.1 Resultados para 15 y 10 mejores reglas por clase. 156

8 IMPLEMENTACIÓN DE NEFCLASS-Q AL ANÁLISIS DE ESPECTROS DE VIBRACIONES 157
8.1 ENTRENAMIENTO Y VALIDACIÓN CON PRUNING SIMPLE 160
8.2 ENTRENAMIENTO Y VALIDACIÓN CON PRUNING DE LAS MEJORES REGLAS 160

8.3 ENTRENAMIENTO Y VALIDACIÓN CON PRUNING DE LAS MEJORES REGLAS POR CLASE 161

CONCLUSIONES **163**

RECOMENDACIONES **164**

BIBLIOGRAFÍA **166**

A LOGICA FUZZY **168**

B REDES NEURONALES **199**

C ARCHIVO RESPUESTA DE LAS LIBRERIAS **216**

LISTA DE FIGURAS

2.1 Agente Inteligente 28
2.2 Componentes de un sistema Inteligente 29
2.3 Modelos cooperativos Neuro-Fuzzy 34
2.4 Sistema híbrido Neuro-Fuzzy 35
2.5 Técnicas de Inteligencia Artificial 36

3.1 Perceptrón 38

4.1 Componentes de NEFCLASS-Q 43
4.2 Estructura del archivo de entrenamiento genérica 44
4.3 Estructura del archivo de validación genérica 45
4.4 Estructura del archivo de clasificación genérico 45
4.5 Estructura archivo de entrenamiento de IRIS 47
4.6 Estructura del archivo de validación de IRIS 47
4.7 Estructura del archivo de clasificación de IRIS 48
4.8 Procesos de NEFCLASS-Q 48
4.9 Creación de la Red 49
4.10 Variable 1 fuzzificada 50
4.11 Proceso creación de reglas 51
4.12 Variable 1. Longitud de sépalo 52
4.13 Variable 2. Ancho de sépalo 52
4.14 Variable 3. Longitud de pétalo 52
4.15 Variable 4. Ancho de pétalo 53
4.16 Regla creada en la Red Neuro-Fuzzy 54
4.17 Proceso de entrenamiento 55
4.18 Conjuntos que tiene pertenencia mayor a cero 56
4.19 Formación de la regla A 57
4.20 Formación de la regla B 57
4.21 Salida de la red neuro-fuzzy 58
4.22 Parámetros de un conjunto fuzzy 59
4.23 Función signo 60
4.24 Conjunto modificado por entrenamiento 61
4.25 Ejemplo entrenamiento 61

5.1 Archivo de entrenamiento 136
5.2 Archivo de validación 136
5.3 Archivo de clasificación 137
5.4 Ventana inicial NEFCLASS-q 137
5.5 Primera página del asistente de creación de red 138

5.6 Creación del directorio del proyecto 139
5.7 Carga de archivos 139
5.8 Definición tipos de archivos cargados 140
5.9 Estructura de la Red 140
5.10 Configuración de la Red Neuro-Fuzzy 141
5.11 Puntos donde actúan los coeficientes de aprendizaje 141
5.12 Visualizar Universos del sistema 142
5.13 Reglas del sistema 143
5.14 Agregar Reglas Fuzzy 143
5.15 Selección proposición 144
5.16 Reglas creadas 144
5.17 Entrenamiento de la red 145
5.18 Validación del entrenamiento 146

6.1 Sistema Inteligente 147
6.2 Estructura del archivo de configuración de parámetros 148
6.3 Archivo de configuración de parámetros de IRIS 149

7.1 Archivo Iris.dat 152

8.1 Espectro de vibración 159

A.1 Conjunto Concreto 173
A.2 Conjunto Fuzzy 173
A.3 Conjunto fuzzy que caracteriza el número de rotaciones por minuto de un disco duro 174
A.4 Conjunto fuzzy Heavy 175
A.5 Universo de discurso para la temperatura de una turbina 175
A.6 Universo de discurso con conjuntos asimétricos 176
A.7 Conjunto fuzzy definido por puntos 177
A.8 Adición de conjuntos fuzzy 179
A.9 Conjunto crisp short y su complemento tall 180
A.10 Intersección de conjuntos fuzzy 180
A.11 Combinación y defuzificación de proposiciones fuzzy 183
A.12 Conjunto fuzzy tall 186
A.13 Conjunto fuzzy heavy 186
A.14 Implicación monótona 187
A.15 Conjuntos fuzzy para variables de entrada y salida 188
A.16 Método de correlación producto 190
A.17 Proceso de agregación y defuzificación 191
A.18 Proceso de descomposición 192
A.19 Defuzificación de un conjunto fuzzy solución 192
A.20 Defuzificación con el método de altura máxima simple 193
A.21 Defuzificación por el método de promedio de máximos 194

A.22 Defuzificación por método de centro máximos 194
A.23 Proceso lógico en un modelo fuzzy tipo singleton 195
A.24 Representación del singleton en una variable de salida 196
A.25 Variable de salida tipo singleton escalada por valores de verdad . . . 196
A.26 Modelado de un sistema fuzzy 197

B.1 Sistema nervioso humano 199
B.2 Celula piramidal 200
B.3 Modelo de una unidad de procesamiento 203
B.4 Función umbral 204
B.5 Función lineal a trazos 205
B.6 Función sigmoidal 206
B.7 Red Neuronal de una capa con feedforward 206
B.8 Red neuronal multicapa 207
B.9 Red neuronal recurrente 208
B.10 Aprendizaje de correción del error 209
B.11 Aprendizaje competitivo 211
B.12 Aprendizaje supervisado 213
B.13 Aprendizaje reforzado 214
B.14 Aprendizaje no supervisado 215

LISTA DE TABLAS

4.1 Conjuntos que poseen mayor pertenencia 53
4.2 Base de conocimiento 56
4.3 Posibles antecedentes de reglas 58
4.4 Activaciones de los conjuntos que componen la regla 58
4.5 Validación para un patrón de datos 62

7.1 Tabla resultados pruning simple 154
7.2 Resultados pruning 15 mejores reglas 155
7.3 Resultados pruning 10 mejores reglas 155
7.4 Resultados para las 15 mejores reglas por clase 156

8.1 Datos del espectro para el patrón de datos 158
8.2 Resultados reconocimiento de espectros con pruning simple 160
8.3 Resultados del reconocimiento de espectros con pruning de las mejores reglas 160
8.4 Resultado reconocimiento de espectros con el pruning mejores reglas por clase 161
8.5 Reglas creadas con los datos de espectros de vibraciones 162
8.7 continuación de las reglas creadas para clasificación de espectros de vibración 163

LISTA DE ANEXOS

ANEXO A. CONCEPTOS DE LOGICA FUZZY... 168
ANEXO B. CONCEPTOD DE REDES NEURONALES ARTIFICIALES........... 198
ANEXO C. ARCHIVO RESPUESTA DE LAS LIBRERIAS................................ 219

RESUMEN

TITULO: HERRAMIENTA DE APLICACIÓN SOFTWARE PARA LA CLASIFICACIÓN DE PATRONES DE DATOS IMPLEMENTANDO UNA ARQUITECTURA NEURO-FUZZY[1].

AUTORES:
FABIO ANDRES CABALLERO ESTEBAN[2]
JULIAN SEBASTIAN SALAMANCA ESPINOSA

PALABRAS CLAVE: NETCLASS-Q, Redes Neuronales, Lógica Fuzzy, Perceptron Fuzzy, Agente Inteligente.

DESCRIPCIÓN:

En los últimos años en la industria se ha venido presentando la necesidad de realizar diagnosticos de maquinaria con base en el estudio de variables (vibraciones, ultrasonido, dinagramas, etc) que definen la condición del equipo, este proceso normalmente lo realizan ingenieros que tiene gran experiencia y conocen los equipos en detalle, para facilitar el diagnótico de maquinaria se plantea en el presente proyecto un software llamado NEFCLASS-Q creado por los autores que es capaz de reconocer patrones de datos con el el fin de clasificar cada uno de ellos.

NEFCLASS-Q un software basado en una arquitectura compuesta por Redes Neuronales y Lógica Fuzzy, capaz al igual que un ser humano, de aprender a partir de ejemplos y expresar el conocimiento adquirido en forma de reglas lingüísticas. La finalidad de NEFCLASS-Q es la clasificación confiable de patrones de datos, pensado como una herramienta en el desarrollo e investigación de los Agentes Inteligentes aplicados a la Industria.

El núcleo de NEFCLASS-Q yace en la arquitectura del Perceptrón Fuzzy, el cual fue creado por los Doctores Ulrike Nauck y Rudolf Kruse en Alemania en al año 1994, esta arquitectura fue implementada por medio de seis librerías en C++ desarrollada por los autores del presente proyecto, las cuales engloban las características y funciones del Perceptron Fuzzy.

Con las librerias desarrolladas se pretende en un futuro proximo crear un sistema automático de diagnostico para implementarlo en los sistemas de levantamiento atificial de crudo.

[1] **Trabajo de grado**

[2] **Facultad de Ingenierías Físico Mecánicas, Escuela de Ingeniería Mecánica, Director: Ing. Jorge Enrique Meneses Florez, Universidad Industrial de Santander**

ABSTRACT

TITLE: APPLICATION SOFTWARE TOOL FOR THE CLASSIFICATION OF DATA PATTERNS IMPLEMENTING A NEURO-FUZZY ARCHITECTURE[3].

AUTHORS:

FABIO ANDRES CABALLERO ESTEBAN[4]
JULIAN SEBASTIAN SALAMANCA ESPINOSA

KEY WORDS: Nefclass, Data patterns, Artificial intelligence, Neuro-Fuzzy, Intelligent agent.

DESCRIPTION:

In recent years the industry has been presenting the need for machinery diagnostics based on the study variables (vibration, ultrasound, dinagramas, etc.) that define the condition of the equipment, this process is done by engineers who normally has great experience and know the equipment in detail, to facilitate machine diagnótico raised in this project a software called NEFCLASS-Q created by the authors that is able to recognize patterns of data in order to classify each of them.

NEFCLASS-Q-based software architecture composed of Neural Networks and Fuzzy Logic, capable as a human being, to learn from examples and express the knowledge acquired in the form of linguistic rules. The purpose of NEFCLASS-Q is the reliable classification of data patterns, intended as a tool in the research and development of intelligent agents applied to the industry.

The core of NEFCLASS-Q lies in the architecture of Perceptron Fuzzy, which was created by Drs Ulrike Nauck and Rudolf Kruse in Germany in the year 1994, this architecture was implemented through six libraries in C + + developed by the authors of this project, which encompass the features and functions of Fuzzy Perceptron.

With the libraries developed is intended in the near future to create an automatic diagnostic systems for deployment atificial oil lifting.

[3]**Graduate work**

[4]Faculty Physical-Mechanics Engineering, Mechanical Engineering School, Director: Jorge Enrique Meneses Florez, Universidad Industrial de Santander

INTRODUCCION

El objetivo del presente trabajo es demostrar el potencial que tiene el campo de la Inteligencia Artificial para solucionar problemas que tenemos hoy en dia a nivel industrial, en el caso del presente proyecto se utilizó una arquitectura Neuro-Fuzzy llamada Perceptrón Fuzzy, la finalidad de esta arquitectura es combinar las tecnicas de las Redes Neuronales Artificiales y los sistemas basados en logica Fuzzy para formar un sistema avanzado mas poderoso combinando las ventajas de cada técnica y dejando atras las desventajas.

El Perceptrón Fuzzy fue creado por el Dr. Rudolf Kruse y el Dr. Ulrike Nauck en Alemania en el año de 1994, y lo utilizaron para desarrollar el software Nefclass. Las necesides que tiene la industria de reconocer formas y patrones como por ejemplo en el caso del analisis de los espectros de vibraciones o el reconocimiento de formas de dinagramas en el campo petrolero, nos llevo a estudiar a fondo la arquitectura del Perceptrón Fuzzy y crear un software llamado NEFCLASS-Q que esta compuesto por una Interfaz grafica que ayudara a entender los procesos de creación, entrenamiento y validación de una Red Neuro-Fuzzy, y por una serie de librerías que estarán destinadas a la creación de Agentes inteligentes, todo esto con el fin de cubrir las necesidades planteadas anteriomente.

Este libro esta compuesto por 8 capitulos donde se explica la arquitectura del perceptrón fuzzy, el codigo fuente de NEFCLASS-Q y los resultados obtenidos.

En el capítulo 1 se presenta la identificación del problema y los objetivos del trabajo de grado, en al capitulo 2 se presenta la importancia de la Inteligencia Artificial y las tecnicas que posee. En los capitulos 3 y 4 se presenta el modelo NEFCLASS y cada uno de los procesos que posee NEFCLASS-Q junto con el codigo fuente del programa. En los capitulos 5 y 6 se explica como utlizar la interfaz grafica y las librerias y por ultimo en los capitulos 7 y 8 se presentas los resultados obtenidos con las base de datos Iris plant y los espectros de vibraciones.

1. ALCANCES DEL PROYECTO

1.1. IDENTIFICACIÓN DEL PROBLEMA

La Inteligencia Artificial es una colección sistemática de habilidades, basada en software, para manejar información proporcionando flexibilidad, ambigüedad, aprendizaje y razonamiento. Esta tiene como objetivo crear sistemas soportados en agentes inteligentes[1], los cuales están compuestos por software, hardware de cómputo, hardware de sensores y hardware de actuadores. El constante desarrollo y la implementación de nuevas tecnologías en la industria exigen el uso de agentes inteligentes para suplir la demanda de personal experto y costoso, estos sistemas deben ser capaces de percibir su entorno mediante sensores, procesar los datos adquiridos y responder en el ambiente por medio de actuadores.

La escuela de Ingeniería Mecánica comprometida con el desarrollo tecnológico de la región y el país, ha venido trabajando en el campo de los agentes inteligentes, específicamente en los sistemas automáticos de diagnóstico, haciéndose cada vez más evidente la necesidad de contar con un software que tenga la capacidad de clasificar patrones de datos. A pesar de que este tipo de software existe en el mercado, cuenta con el inconveniente de que es difícilmente adaptable para los requerimientos de desarrollo e investigación de los agentes inteligentes o en la escuela.

Para la construcción de esta herramienta se dispone de diversas técnicas de inteligencia artificial[2] como lo son las redes neuronales, los sistemas basados en conocimiento y los sistemas híbridos; dentro de los últimos los sistemas Neuro-Fuzzy, que cumplen adecuadamente el concepto de agente inteligente y han sido utilizados ampliamente en la aplicación específica de reconocimiento de patrones.

---

[1] Véase la sección 2.1

[2] Véase Anexo A y B

1.2. JUSTIFICACIÓN

La investigación y el desarrollo de nuevas tecnologías en las Universidades son importantes para el progreso tecnológico de la región y por esto desde el año 1994 en la escuela de Ingeniería Mecánica se han elaborado proyectos de grado en el campo de la Inteligencia Artificial enfocado a tres tópicos en particular:

1. Diseño óptimo
2. Control Automático
3. Diagnóstico

A pesar de esto, el interés que se presenta por esta área(inteligencia artificial) al interior de la escuela de ingeniería mecánica, no es el adecuado, pensando en la aplicabilidad que tiene esta en nuestro campo. Teniendo en cuenta los trabajos realizados en el campo, la potencialidad que representan y la necesidad de construir e implementar agentes inteligentes en el área de sistemas automáticos de diagnóstico, se hace indispensable tener a disposición una herramienta propia, fácilmente adaptable, de alto desempeño y que cumpla los requerimientos necesarios para el desarrollo de los agentes inteligentes.

1.3. OBJETIVOS

1.3.1. Objetivo general

Contribuir con la misión de la Universidad Industrial de Santander de formar personas con capacidad de investigación e innovación, desarrollando herramientas computacionales para aportar a los procesos de crecimiento intelectual y tecnológico de la región y el país.

1.3.2. Objetivos específicos

- Desarrollar una herramienta de aplicación software para la clasificación de patrones de datos utilizando una arquitectura Neuro-Fuzzy. Este software permitirá:

  - Crear un sistema Neuro-Fuzzy con la estructura adecuada que permita el entrenamiento y aprendizaje para un problema dado por el usuario.
  - Generar una base de conocimiento en forma de reglas “si-entonces” expresada en términos lingüísticos, fácilmente interpretables por el ser humano.
  - Dar la flexibilidad al usuario de definir parámetros iniciales del sistema Neuro-Fuzzy como las ratas de aprendizaje, base de conocimiento inicial, número de conjuntos fuzzy por variable, número de épocas de entrenamiento y máximo número de reglas.
  - Tener la posibilidad de trabajar con un máximo de 50 variables de entrada en la red.
  - Generar un máximo de 500 reglas de la forma a “si-entonces” usando términos lingüísticos.
  - Hacer la clasificación con un error máximo del 10 %.

- Construir una interfaz gráfica con el fin de facilitar a la interacción entre la herramienta y el usuario.
- Redactar el manual de usuario, para un mejor uso de la herramienta.
- Validar el software con la base de datos Iris Plant.

1.4. JUSTIFICACIÓN DE LA SOLUCIÓN

Actualmente en el mercado existen numerosas herramientas enfocadas al diseño y creación de software, que de una u otra manera facilitan estos procesos, por lo tanto se hace necesario hacer un análisis de alternativas en busca de que el software a desarrollar tenga la calidad requerida para poder ser implementado en el ámbito industrial. Además existen diferentes modelos de arquitecturas *Neuro-Fuzzy* en los cuales podemos centrar nuestra investigación para la implementación del algoritmo.

1.4.1. Alternativas de diseño

Los modelos de sistemas aunque escasos son muy eficientes, podemos encontrar aplicaciones que van desde la predicción hasta el control automático pasando por el reconocimiento y clasificación de patrones, en campos como la medicina, la industria del petróleo, la ingeniería, etc. Actualmente existen claros ejemplos de software de aplicación en cada una de estas áreas, pero lamentablemente no es posible acceder a las estructuras de todos ellos, y si es posible, muy difícilmente pueden ser editados para la adaptación a otro tipo de aplicación distinta para la que fueron diseñados.

1.4.1.1. Arquitectura ANFIS

ANFIS (Adaptive Neuro-Fuzzy Inference Systems) es una arquitectura utilizada comúnmente en sistemas de identificación y control, que partiendo de las mediciones de entrada y de salida del proceso elabora un sistema de inferencia difusa sobre la base de una red adaptable *neuro-fuzzy*, facilitando este proceso por medio de una interfaz gráfica. Este usa un algoritmo de aprendizaje híbrido, con una arquitectura que consta de una capa de entrada, una capa de salida y tres capas ocultas.

Ventajas:

Posee una arquitectura robusta que le permite funcionar como clasificador y aproximador universal. Utiliza un procedimiento de aprendizaje para el cual existen dos alternativas de algoritmo, acomodándose a la necesidad del problema. El sistema de inferencias generado puede ser utilizado con el simulink de matlab.

Desventajas:

No existe acceso al código fuente. Debido a la robustez del algoritmo se hace más complicado la implementación y reutilización en la herramienta para el desarrollo de agentes inteligentes.

1.4.1.2. Arquitectura Perceptrón fuzzy (NEFCLASS)

NefClass (Neuro-Fuzzy Classification) es un software usado para derivar reglas difusas a partir de un conjunto de datos que pueden ser separados en distintos grupos de clases. La tarea del modelo NefClass es descubrir estas reglas y aprender la forma de la función de pertenencia que determina la clase correcta o categoría de un patrón de entrada dado. La topología del perceptrón usado en NefClass consiste en una variante del perceptrón fuzzy[3], contando con una capa de entrada, representando las variables del modelo, una única capa oculta, simbolizando las reglas lingüísticas y una capa de salida que hace referencia a las clases a diferenciar.

Ventajas:

El modelo es "simple" en comparación con otras arquitecturas, lo cual facilita su codificación y posterior uso. La arquitectura es especializada para la clasificación, este modelo a diferencia de otras arquitecturas hace que las reglas de clasificación sean fácilmente interpretables. Existe un precedente al interior de la escuela de ingeniería mecánica que demuestra la aplicación de este algoritmo (usando NefClass) en la clasificación de patrones de datos, mostrando excelentes resultados.

Desventajas:

En este modelo se sacrifica la exactitud para mejorar la legibilidad de la clasificación realizada. El código fuente existente es difícilmente adaptable para la aplicación en los agentes inteligentes debido a que fue diseñado como software de aplicación únicamente.

1.4.1.3. Arquitectura Perceptrón fuzzy (NEFCON)

NefCon (Neuro-Fuzzy Control) es un software de aplicación que implementa una variación del perceptrón difuso, en este caso la especialización del algoritmo busca crear un set de conjuntos difusos para afinar un controlador *fuzzy*, es por esto que este modelo no es aplicable para la clasificación de datos.

1.4.1.4. Arquitectura Perceptrón fuzzy (NEFPOX y NFIDENT)

NefProx (Neuro-Fuzzy Function Approximation) es otro software de aplicación que implementa otra variación del perceptrón difuso aplicándolo a la aproximación de funciones, esta arquitectura tampoco es aplicable a la clasificación de patrones de datos.

[3]Véase la sección 3.1

1.4.2. Alternativas de desarrollo

Hoy en día las opciones de las que disponemos, en cuanto a lenguajes de programación, para hacer el desarrollo y creación de software son muy amplias, pasando desde lenguajes de programación tan flexibles como *JAVA* hasta los complejos lenguajes de bajo nivel. Entre las alternativas para el desarrollo de la herramienta fueron seleccionados los más conocidos e implementados en la industria del software.

1.4.2.1. Lenguaje de programación C

C es un lenguaje de programación de propósito general que ofrece economía sintáctica, y de nivel medio ya que combina los elementos del lenguaje de alto nivel con la funcionalidad del ensamblador.

Ventajas:

Es altamente transportable. Es muy flexible. Genera código muy eficiente.

Desventajas:

Es muy poco modular, y es difícil la interpretación del código si es hecho por otras personas. Es un lenguaje que se basa en la programación estructurada y carece de polimorfismo en forma de sobrecarga, es decir el código generado es poco reutilizable.

1.4.2.2. Lenguaje de programación M

Matlab es un software matemático que ofrece un IDE (Intergrade Development Environment) con un lenguaje de programación propio (lenguaje M).

Ventajas:

Está disponible para las plataformas Unix, Apple, Windows y Mac Os-X. Cuenta con una amplia variedad de funciones pre-programadas que facilitan en gran medida la tarea del programador. Tiene incorporado un módulo especializado en el entrenamiento de redes neuronales y además posee herramientas enfocadas al manejo de conjuntos difusos.

Desventajas:

Debido a que los programas hechos en matlab requieren de un intérprete para poder funcionar, su velocidad en el procesamiento disminuye de forma significativa. Para desarrollar programas en este lenguaje (de forma legal), es necesario disponer de una licencia estándar la cual es sumamente costosa. El lenguaje M está enfocado a la programación estructurada. Carece de polimorfismo, por lo que la reutilización de código es tediosa.

1.4.2.3. Lenguaje de programación *JAVA*

Java es un lenguaje de programación orientado a objetos y fue creado con el propósito de que pudiera funcionar independiente de la plataforma en que se ejecute.

Ventajas:

Es simple al eliminar la complejidad de los lenguajes como C. Es altamente portable. Permite crear programas modulares y códigos reutilizables.

Desventajas:

Dado que los programas hechos en Java requieren de un intérprete, los procesos son más lentos comparados con otros lenguajes.

1.4.2.4. Lenguaje de programación C++

C++ es un lenguaje de programación que fue creado con la intención de ser una extensión de C, añadiendo características que le permitieran lograr una mejor abstracción de los problemas.

Ventajas:

Es un lenguaje de programación orientado a objetos. Cuenta con polimorfismo, herencia y sobrecarga, lo cual hace que el código desarrollado sea altamente reutilizable. Existe la posibilidad de redefinir los operadores (sobrecarga de operadores). Se pueden crear nuevos tipos de datos que se comporten como tipos fundamentales. Los programas hechos en C++ son de alto desempeño, ya que no requieren de ningún intérprete para funcionar. Debido a su gran implementación en el ámbito informático, existe numerosa documentación que facilita la programación. Pueden desarrollarse programas en múltiples plataformas y con la misma eficiencia.

Desventajas:

El manejo de memoria dinámica requiere de cierta habilidad del programador. Iniciarse en la programación con C++ es un poco complicado.

1.4.3. Alternativa seleccionada

En base a los argumentos anteriormente descritos y los objetivos planteados, se busca contar con una arquitectura que sea simple pero eficiente, esto con el fin de que sea fácilmente codificable además de que facilite futuros usos para su edición y mejoramiento en busca de que la Escuela de Ingeniería Mecánica de la Universidad Industrial de Santander cuente con una herramienta potente y versátil a la hora de

realizar futuras investigaciones en el desarrollo de los agentes inteligentes. Por ende se implementara la arquitectura de NefClass derivada del modelo de perceptrón *fuzzy*. Pensando en la implementación de software libre para el desarrollo de la herramienta como uno de los objetivos del proyecto, además sustentados en la idea de las facilidades presentadas por una programación orientada a objetos, e igualmente en busca de una alta velocidad de procesamiento independiente de la plataforma, se seleccionó C++ como el lenguaje de programación a utilizar en el desarrollo de la herramienta. INTELIGENCIA ARTIFICIAL

La Inteligencia Artificial se define como una colección sistemática de habilidades para ser implementadas en Agentes racionales no vivos, de manera más especifica la Inteligencia Artificial es la disciplina que se encarga de proporcionar las herramientas necesarias para la construcción de algoritmos que al ser ejecutados sobre un entorno produce acciones o resultados que maximizan una medida de rendimiento determinada basándose en la secuencia de entradas percibidas y el conocimiento almacenado.

Existen diferentes tipos de conocimiento y medios de representación del mismo, así como se distinguen varias formas de procesos validos para obtener resultados racionales. La combinación de estos dos factores junto con la capacidad de percibir y actuar formas lo que se conoce como Agente Inteligente.

2. AGENTES INTELIGENTES

Un sistema Inteligente es cualquier entidad que puede percibir su entorno a través de sensores, procesar tales percepciones por medio de habilidades programadas usando técnicas de Inteligencia Artificial Figura 2.1, para responder sobre un entorno de manera racional usando actuadores, entendiéndose la racionalidad como una capacidad humana que permite pensar, evaluar y actuar conforme a ciertos principios de optimización y consistencia, para satisfacer algún objetivo o finalidad.

Figura 2.1: Agente Inteligente

Modificado de http://jhanlos.blogspot.com/2009/11/agentes-inteligentes.html

Los componentes básicos de un Agente Inteligente incluye los siguientes elementos :

- Software
- Hardware computacional
- Sensores
- Actuadores

El software lo componen los algoritmos y métodos para el procesamiento de datos y almacenamiento del conocimiento adquirido durante la existencia del Agente, el hardware computacional es la interfaz de comunicación entre el software y la arquitectura física, esta última dispone de los elementos perceptivos y actuadores que interactúan con el entorno en el que se encuentra el agente.

Se puede encontrar una gran variedad de aplicaciones en la que se ven involucrados los agentes inteligentes, algunos ejemplos son:

- Un software *softbot* que simula a una persona en un juego de computadora, tal como un jugador de ajedrez, un jugador de fútbol o un conductor de carreras de automóviles, etc.

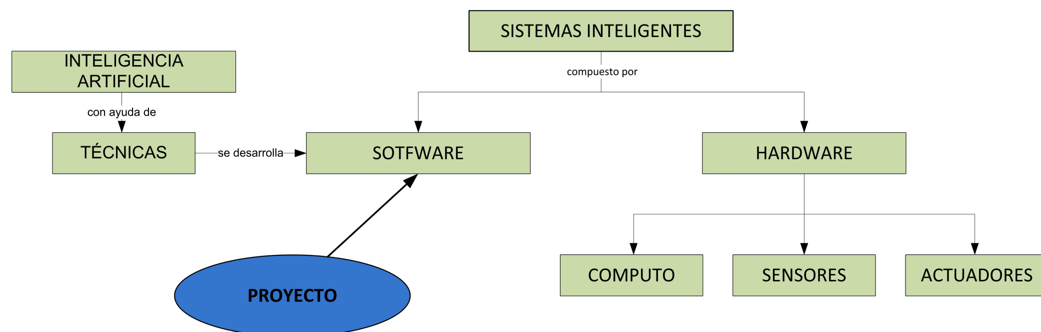
- Un sistema de diagnóstico automático que capta los espectros de frecuencia vibratoria en la caja de rodamientos de un equipo de bombeo y predice el tipo de falla que se presentara y emite una señal indicando estos datos al operario.

- Una computadora especializada que controla un helicóptero en maniobras peligrosas para un hombre.

- Un robot de comportamiento variable auto regulado (ya sea que su comportamiento sea determinado por software o incorporado directamente en la electrónica).

El presente proyecto aborda uno de los grandes retos en proceso de desarrollo de Agentes Inteligentes, proyectándose como la base computacional del sistema de procesamiento de datos en un Agente Inteligente, y enfocado al desarrollo de un sistema automático de diagnóstico.

Figura 2.2: Componentes de un sistema Inteligente



Fuente: Autores

Un sistema automático de diagnóstico cuenta con la capacidad de reconocer distintos estados específicos dados por el entorno y definidos por un número determinado de variables, en este orden de ideas, la herramienta computacional debe ser capaz de distinguir patrones en las variables que definen un estado.

Existen gran variedad de técnicas con las que se puede desarrollar esta habilidad y que proporcionan distintas ventajas a la hora de ser usadas.

2.1. TÉCNICAS DE INTELIGENCIA ARTIFICIAL

Existen múltiples métodos usados en la Inteligencia artificial, para desarrollar los aspectos fundamentales de un sistema inteligente, abarcando así el aprendizaje, el conocimiento generado y la generación de respuestas lógicas. Cada uno de ellos proporciona ventajas competitivas respecto a los demás modelos, pero a su vez presentan desventajas con los mismos. Las técnicas más representativas son:

- Redes Neuronales Artificiales
- Lógica Fuzzy
- Algoritmos Genéticos
- Sistemas Híbridos

2.1.1. Redes Neuronales Artificiales

Es un sistema formado por una cantidad determinada de elementos de procesamiento llamados nodos o neuronas, interconectados en una arquitectura inspirada en la estructura del cerebro. Las ventajas que se tienen con esta técnica son:

- Adquieren el conocimiento a partir de ejemplos.
- El conocimiento se almacena de un forma descentralizada, es decir, que si desaparecen algunos nodos de la red no se verá afectado en gran medida el conocimiento almacenado en la misma.
- No requiere de una base de conocimiento, ya que esta es adquirida durante el aprendizaje.
- No requiere de modelos matemáticos para el proceso.

Por otro lado existen ciertos aspectos en forma de desventajas:

- Se comportan como una caja negra, es decir, los procesos al interior de la misma son inciertos.
- Después del entrenamiento de la Red, no se puede extraer le conocimiento.
- Si se dispusiera de la configuración final (después del entrenamiento de la red) no sería posible por un ser humano interpretar tal información.

Si bien este tema puede prestarse para pensar que el trabajo allí realizado solo es aplicable a las ciencias computacionales, existen infinidad de aplicaciones en

campos de la ingeniería[1] , la medicina[2], la economía, etc, mostrando en cada uno de ellos excelentes resultados.

2.1.2. Sistemas basados en Lógica Fuzzy.

La lógica Fuzzy o difusa es un área de la soft computing que permite al software manejar variables con incertidumbre. Estos sistemas son compuestos por inferencias Fuzzy que contiene reglas sintácticas de la forma **SI** *predicado a* y *predicado b* ... entonces *Consecuente*, formando estas lo que se conoce como base de reglas. Los sistemas basados en lógica fuzzy imitan la forma en que tomas decisión los seres humanos con la ventaja de ser mucho más rápidos. Las ventajas presentes en estos sistemas son:

- No requiere de modelos matemáticos para el proceso.
- Permite usar conocimiento previo (Reglas de la forma Si-Entonces).
- Gracias al uso de las Reglas sintácticas se facilita la interpretación por el ser humano.

Aún así existen grandes problemas a la hora de generar los modelos fuzzy.

- No pueden tener procesos de entrenamiento, es decir no pueden aprender.
- No existen métodos formales de optimización de conjunto fuzzy.
- El sistema funciono solo para el entorno en el que fue creado

Una de las más populares aplicaciones de los sistemas fuzzy en el campo de la Ingeniería se encuentra relacionado principalmente con el control de procesos industriales complejos.

2.1.3. Algoritmos genéticos

El algoritmo genético es una técnica de búsqueda basada en la teoría de la evolución de Darwin. Los algoritmos genéticos son métodos adaptativos que pueden usarse para resolver problemas de búsqueda y optimización. Están basados en el proceso genético de los organismos vivos. Las ventajas son las siguientes:

- Operan de forma simultánea con varias soluciones, en vez de trabajar de forma secuencial como las técnicas tradicionales.

[1]En la Escuela de Ingeniería Mecánica de la Universidad Industrial de Santander se desarrollo el proyecto “Aplicación de la tecnología de Redes Neuronales a la fase de diagnóstico del Mantenimiento predictivo de equipo Industrial” realizado por Mario Alfonso Mantilla y William Francisco Murillo dirigido por el Profesor Jorge Enrique Meneses.

[2]Diseño e Implementación de un sistema basado en Redes Neuronales artificiales para la caracterización de células malignas en fluido pleural realizado por Juan Caviedes, Adrian Rodriguez dirigido Olga Alvarez y Víctor Abaunza.

- Usan operadores probabilísticos, en vez de los típicos operadores determinísticos de las otras técnicas.

Las desventajas son las siguientes:

- Pueden tardar mucho en dar una solución, o no encontrarla, dependiendo en cierta medida de los parámetros que se utilicen tamaño de la población, número de generaciones, etc.

- Pueden converger prematuramente debido a una serie de problemas de diversa índole.

2.1.4. Sistemas Híbridos Inteligentes

Los sistemas híbridos inteligentes denotan a los sistemas software que emplean en paralelo una combinación de modelos de inteligencia artificial, métodos y técnicas de éstos subcampos como:

- Sistemas Neuro-Fuzzy.

- Redes neuronales evolutivas

- Sistemas Genetic-Fuzzy-Neural

- Sistemas difusos genéticos.

- Aprendizaje de algoritmos genéticos difusos (Genetic algorithm fuzzy reinforcement learning, GAFRL)

Gracias a los antecedentes que hemos tenido no solo en la Escuela de Ingeniería Mecánica sino en el mundo y a las ventajas que tienen los sistemas Neuro-Fuzzy, podemos decir que estos sistemas nos pueden proporcionar las características necesarias para el desarrollo de la herramienta computacional.

2.2. SISTEMAS NEURO-FUZZY

Los sistemas Neuro-Fuzzy nacen de la necesidad de solucionar los problemas que presentan los sistemas basados en Lógica Fuzzy que son:

- Requiere conocimiento previo explícito en forma de reglas Si-entonces y conjuntos Fuzzy.

- Requiere una sintonización manual de los conjuntos Fuzzy, lo que lleva tiempo y es propenso al error.

Estos problemas se solucionan mediante un proceso de aprendizaje automático de las reglas y de los conjuntos Fuzzy. Este proceso de aprendizaje lo pueden dar las Redes Neuronales Artificiales ya que una de sus ventajas es el aprendizaje a partir de ejemplos. En conclusión la intención de los sistemas Neuro-Fuzzy es crear o mejorar un sistema Fuzzy apoyado por las redes Neuronales Artificiales, o mejorar el proceso de aprendizaje de una Red Neuronal Artificial a partir de la Lógica Fuzzy presentando las siguientes características:

- Acortar el tiempo de aprendizaje usando conocimiento previo.
- Habilidad de aprendizaje a partir de ejemplos por medio del entrenamiento del sistema.
- Proporciona medios para interpretar los resultados con criterios ambiguos, reglas de la forma Si-Entonces.
- Resultados más eficientes en períodos más cortos.

2.2.1. Modelos Neuro-Fuzzy

Existen dos modelos Neuro-Fuzzy que nos pueden ayudar en el proceso de la clasificación de patrones de datos que son:

- Modelos cooperativos Neuro-Fuzzy.
- Modelos híbridos Neuro-Fuzzy.

2.2.1.1. Modelos cooperativos Neuro-Fuzzy

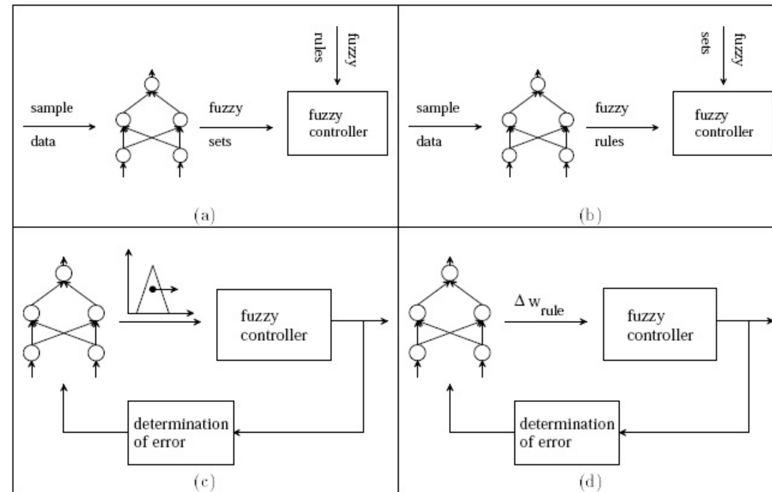
Los modelos cooperativos Neuro-Fuzzy es una posible combinación de las Redes Neuronales y Sistemas Fuzzy, que usa los modelos de manera independiente. La combinación se debe a que la red neuronal aprende u optimiza los diversos parámetros de un controlador fuzzy, antes de ser usado (offline), o mientras esta en operación (online).

En la figura 2.3 se presenta los diferentes tipos de modelos cooperativos Neuro-Fuzzy, En la figura 2.3a la Red Neuronal proporciona las funciones de pertenencia al sistema fuzzy, esto puede ser hecho por medio de un aprendizaje especial de parámetros o usando varias Redes Neuronales para aproximar la función de pertenencia directamente. Los conjuntos fuzzy son aprendidos offline y son usados con reglas fuzzy predefinidas para implementar el sistema fuzzy. Una forma de aprender los parámetros de una función de pertenencia es descrito en el libro de Nomura[3]

En la figura 2.3b la Red Neuronal proporciona las reglas lingüísticas de unos datos de entrenamiento, esto se realiza generalmente por medio de algoritmos de

[3] Nomura, I. Hayashi and N.Wakami (1992). A Learning Method of Fuzzy Inferences Rules by Descent Method. In proc. IEEE Int. Conf of Fuzzy Systems of 1992. Pages 203-210. San Diego.

Figura 2.3: Modelos cooperativos Neuro-Fuzzy



Fuente: NAUCK, Detleft; KRUSE, Rudolf. Choosing Appropriate Neuro-Fuzzy Model. Technical University of Braunschweig, Department of Computer Science, Germany.

agrupamiento, el entrenamiento tiene que ser hecho antes de la ejecución y los conjuntos fuzzy tienen que ser definidos de otra forma.

En la figura 2.3c la Red Neuronal adapta los parámetros del conjunto fuzzy durante la ejecución, quiere decir, mientras el sistema fuzzy está en funcionamiento, por esto se debe conocer las reglas y los conjuntos fuzzy iniciales, además se debe definir una medida del error que permita guiar el proceso de aprendizaje. Si se dispone de una labor fija de aprendizaje, este modelo también puede aprender bajo las características offline.

En la figura 2.3d las Redes Neuronales aprenden factores de pesos aplicados a las reglas Fuzzy en un modo online u offline. Este tipo de pesos son usualmente interpretados teniendo en cuenta la importancia de una regla, por lo tanto generan las modificaciones en la salida del sistema. Para esto las reglas fuzzy y los conjuntos fuzzy deben ser conocidos anticipadamente. La semántica para este tipo de pesos no es clara, la variación de un peso en una regla puede modificar los conjuntos fuzzy, esto conduce a conjuntos fuzzy anormales e idénticos valores son representados en diferentes formas y diferentes reglas.

Estos tipo de sistemas Neuro-Fuzzy son usados en Japón para el consumo de productos que generalmente se adaptan a los deseos del cliente.

2.2.1.2. Modelos híbridos Neuro-Fuzzy.

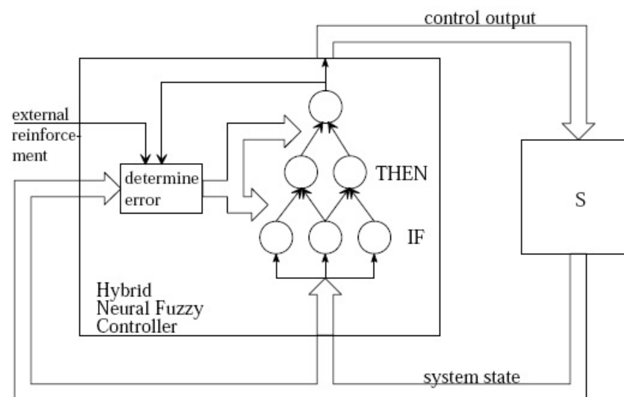
El modelo de sistemas híbridos Neuro-Fuzzy es un sistema fuzzy que puede interpretarse como una Red Neuronal especial o un sistema fuzzy que emplea el procedimiento de aprendizaje de las redes neuronales. En la figura 2.4 es mostrado un sistema híbrido Neuro-Fuzzy genérico. La ventaja de este modelo radica en la

consistencia de su arquitectura que permite establecer una comunicación entre los diferentes modelos. La idea de un modelo híbrido es la interpretación de la base de reglas Fuzzy en términos de una Red Neuronal, por ejemplo, los conjuntos fuzzy son interpretados como los pesos de la red y las reglas, variables de entrada y variables de salida son interpretadas como neuronas.

El algoritmo de aprendizaje en la Red Neuronal da como resultado un cambio de la arquitectura, como la adaptación de los pesos, la creación o eliminación de conexiones, estos cambios pueden ser interpretados en ambos términos, Redes Neuronales y Sistema Fuzzy. Este último aspecto es muy importante ya que la Red no se comporta como una caja negra porque el algoritmo de aprendizaje presenta un resultado óptimo, ya que el conocimiento puede ser interpretado en forma de reglas fuzzy.

Tomando en cuenta las características de los modelos Neuro-fuzzy, el más adecuado para la clasificación de patrones de datos, es el modelo híbrido Neuro-Fuzzy porque nos ofrece la interpretabilidad de las reglas generadas en el entrenamiento de la red, la posibilidad de agregar conocimiento previo al sistema para acortar los tiempos de entrenamiento y una sintonización automática de los conjuntos fuzzy. Actualmente existe un software especializado para la clasificación de patrones de datos llamado NEFCLASS. Este software fue elaborado con base en un modelo híbrido Neuro-Fuzzy por Detleft Nauck (Technical University of Braunschweig, 1997). El modelo NEFCLASS está basado en el modelo del perceptrón fuzzy de Detleft Nauck y Rudolt Kruse. La razón por la que no utilizamos el software NEFCLASS fue porque no era fácilmente adaptable para la creación de agentes inteligentes debido a que su código fuente es difícilmente reutilizable, no hay documentación acerca de las funciones principales del código y la interfaz es poco amigable para el usuario

Figura 2.4: Sistema híbrido Neuro-Fuzzy



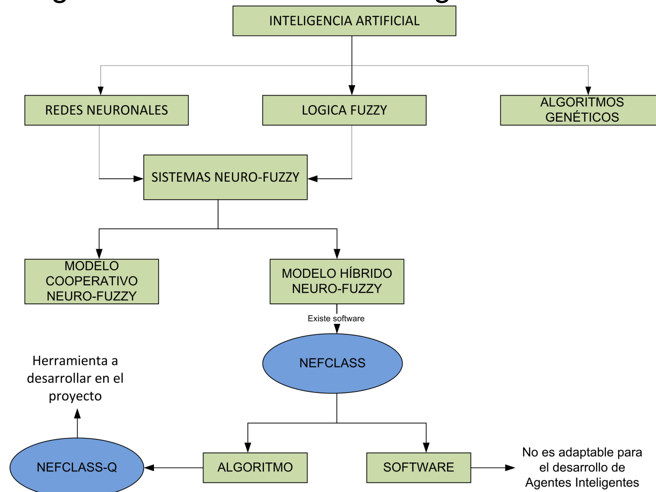
Fuente: NAUCK, Detleft; KRUSE, Rudolf. Choosing Appropriate Neuro-Fuzzy Model. Technical University of Braunschweig, Department of Computer Science, Germany.

elaborado como una aplicación DOS.

Existe un antecedente en la Universidad Industrial de Santander, de la implementación del Software NEFCLASS en clasificación de patrones de datos, el proyecto se llamó “Sistema Neuro-Fuzzy: Prospectivas de la aplicación en la detección de fallas en equipos industriales rotativos”. realizado por Joaquín Enrique Guerrero y Carlos Eduardo Rodriguez, dirigido por el Profesor Jorge Enrique Meneses. En este proyecto se clasificaron espectros de vibraciones, obteniendo mejores resultados comparado con los que se obtuvieron con las Redes Neuronales Artificiales. Una de las recomendaciones de este proyecto fue el desarrollo y la implementación de una herramienta software para clasificación de patrones de datos propia.

Por estas razones se decidió crear un software para clasificación de patrones de datos basado en el modelo NEFCLASS. Para este objetivo es necesario primero entender la arquitectura del modelo NEFCLASS y su respectivo algoritmo. En la figura 2.5 está el resumen de las técnicas disponibles de Inteligencia artificial.

Figura 2.5: Técnicas de Inteligencia Artificial



Fuente: Autores

3. MODELO NEFCLASS

El modelo NEFCLASS es usado para determinar la categoría correcta de un patrón de datos, es una propuesta que construye un sistema fuzzy propagando patrones de datos a través de la red y por medio de un algoritmo de aprendizaje se calcula los parámetros locales y sus modificaciones. Existen diversos métodos que son capaces de hacerlo, pero una de las principales características del modelo es la interpretabilidad de la clasificación, mayor rapidez en la clasificación, y en lo que se refiere al algoritmo de aprendizaje, el modelo permite que el usuario pueda influir en dicho proceso.

Por lo tanto el objetivo principal de NEFCLASS es crear un clasificador legible, que proporcione una exactitud aceptable ya que normalmente estas dos características no pueden estar en un sistema Neuro-Fuzzy, en el sentido que lo que gana una lo pierde la otra.

Un sistema Neuro-Fuzzy interpretable puede mostrar las siguientes características:

1. Pocas reglas significativas con pocas variables en sus antecedentes.
2. Pocos conjuntos en cada variable.
3. Términos lingüísticos que son representados por idénticos conjuntos fuzzy.
4. Normalmente son usados los conjuntos fuzzy, o aún mejor números fuzzy o intervalos fuzzy.

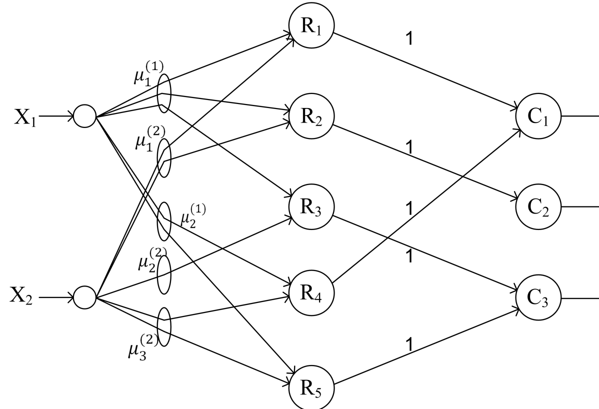
Estas características implican unas restricciones en la creación de un sistema Neuro-Fuzzy, podemos variar los parámetros del mismo con el fin de buscar alta interpretabilidad del conocimiento generado o una elevada exactitud, pero teniendo en cuenta que el aumento de una desfavorece la otra, en donde cada una de ellas cumple una función específica para determinada aplicación.

3.1. ARQUITECTURA DEL MODELO NEFCLASS

La figura 3.1 esquematiza el flujo de datos paralelo a través de la estructura, tanto para el proceso de entrenamiento como para la clasificación; este sistema no es propiamente una red neuronal, es una red híbrida, sí fuese una red neuronal convencional las ventajas que conlleva el uso de la lógica fuzzy se perderían, así mismo solo si se considera como un sistema difuso no existiría la adaptabilidad ni el aprendizaje por medio de ejemplos.

En la figura 3.1 muestra el modelo NEFCLASS que puede ser visto como una Red Neuronal de 3 capas *feedforward* donde cada una de las capas cumple una función:

Figura 3.1: Perceptrón



Fuente: Autores

CAPA DE ENTRADA: Esta primera capa representa los nodos fuente y para fines prácticos funciona como módulos de procesamiento. Esta capa de entrada es la encargada de recibir cada uno de los patrones de datos, donde cada unidad de procesamiento de la capa de entrada recibe un dato numérico, y su activación o respuesta es igual a la entrada externa, y lo conecta a las respectivas unidades de procesamiento llamadas reglas.

CAPA OCULTA: O capa de reglas, es la encargada de realizar las inferencias lingüísticas, almacenan las reglas, es decir conectan el antecedente con el consecuente, la activación de estos nodos es dado por un operador fuzzy.

TERCERA CAPA: O capa de clases, es la que representa las diferentes opciones a la que puede pertenecer un patrón de datos, y se encarga de hacer el proceso de calculo del error.

Remitiéndonos a la figura 3.1, μ_j^i representa los pesos sinápticos $w\left(x_i, R_k\right)$ donde j representa la variable de entrada a la unidad de procesamiento de la primera capa, y la i representa el número del conjunto para la variable j, En otras palabras cada variable tiene un universo de discurso donde es definida de manera imprecisa por medio de los conjuntos fuzzy.

El modelo NEFCLASS usa pesos compartidos en algunas de sus conexiones, y de esta manera asegura que para cada valor lingüístico exista una única representación fuzzy. Por lo tanto, no puede suceder que dos conjuntos que son idénticos al comienzo se desarrollen de forma diferente en el proceso de aprendizaje. Las conexiones que comparten pesos, como $\mu_1^{(1)}$ de la figura 3.1, siempre provienen de la misma unidad de entrada, ya que una etiqueta siempre tiene el mismo significado para una variable de entrada, pero no necesariamente para todas las demás.

Todos los pesos de las conexiones (conjuntos fuzzy) que salen de una unidad de procesamiento de la capa de entrada son las que definen el lenguaje aproximado

o inexacto de la correspondiente variable que representa el nodo de entrada en el universo de discurso, entonces $\mu_1^{(1)}$ $\mu_2^{(1)}$ $\mu_3^{(1)}$ son los conjuntos que están dentro del universo de discurso de la variable χ_1.

Continuando con la figura 3.1, R_k representa las reglas, donde a esta le puede llegar una o más conexiones de la capa de entrada, la activación o respuesta de la neurona de la capa oculta, estará dada por:

$$a_R^{(p)} = \underset{x \in U_1}{min} \{W(x, R)(a_x^{(p)})\}$$

En otras palabras la salida de R_k está determinada por el peso de la conexión que tenga el valor más pequeño, este valor depende de la forma que tenga el conjunto fuzzy de cada conexión y del patrón que se está propagando a través de la red. Es así como dependiendo de las conexiones que se tengan en la arquitectura, está implícita la regla semántica, pero hay que tener en cuenta que por cada patrón de datos que se esté propagando se puede activar más de una regla.

$w\,(R_k, C_m)$ es la conexión de la regla R_k a la unidad de salida C_m. Por razones semánticas los pesos de estas conexiones son valores fijos, y pueden tener el valor de 1 o de 0 (si la conexión no existe). Cada neurona de la capa escondida es conectada solamente a una unidad de salida y la función de activación de las neuronas de salida es un operador fuzzy. La activación de una neurona de salida está dada por la siguiente ecuación.

$$a_c^{(p)} = \underset{R \in U_2}{max} \{a_R^{(p)}\}$$

Es decir, si la neurona de salida esta interconectada con más de una Regla, la activación está determinada por la Regla conecta a dicha neurona de salida que tenga mayor activación. La principal característica que tiene una neurona de salida es la activación que pude ser 1 o 0, las activaciones de estas determinaran la clasificación del patrón propagado.

Una red Neuro-Fuzzy puede ser creada con el modelo NEFCLASS a partir de conocimiento previo, o a partir de una base de reglas vacías, que es llenada en el proceso de aprendizaje con datos. Por cada variable de entrada se debe definir cuantos conjuntos fuzzy van a constituir el universo de discurso con sus respectivos límites, igualmente se puede especificar el número máximo de reglas a crear en la capa escondida de la arquitectura, con la condición de que por cada clase debe existir al menos una regla. La función de pertenencia utilizada es de forma triangular y es definida por 3 parámetros:

$$\mu : \mathbb{R} \to \mu(x) = \begin{cases} \frac{x-a}{b-a} & si\ x \in\ [a, b) \\ \frac{c-x}{c-b} & si\ x \in\ [b, c] \\ 0 & en\ otro\ caso \end{cases}$$

Adicionalmente los conjuntos de los extremos son representados como la mitad de un trapecio (cada uno).

3.2. CONFIGURACIÓN DEL MODELO

Considérese un sistema NEFCLASS con la siguiente configuración:

1. n unidades de entrada representadas por $U_1 = \{x_1, x_2, ..., x_n\}$. Donde U_1 representa la capa de entrada, n el número de variables de entrada y x la neurona de la capa de entrada.

2. Un número $k \leq k_{max}$ de reglas $U_2 = \{R_1, R_2, ..., R_K\}$. Donde U_2 representa la capa escondida, k el número de reglas y R la regla o neurona de la capa escondida .

3. m unidades de salida de la capa $U_3 = \{C_1, C_2, ..., C_m\}$. Donde U_3 representa la capa de salida, m el número de clases y C la neurona de salida.

4. Cada conexión entre las unidades $x_i \in U_1$ y $R_j \in U_2$ tiene un peso que es un conjunto fuzzy, el cual es etiquetado con el término lingüístico $\mu_{jr}^{(i)}(jr \in \{1, 2, ..., q_i\})$

5. Los pesos de las conexiones sinápticas $W(R, C) \in \{1, 0\}$ para todo $R \in U_2$ y $C \in U_3$. Es decir los pesos entre las unidades de procesamiento de la capa escondida y la capa de salida pueden ser 1 o 0.

6. Si denotamos a $L_{x,R}$ como todas las etiquetas lingüísticas de las conexiones entre las $x \in U_1$ y $R \in U_2$ para todas $R, R^{'} \in U_2$ se cumple que: $(V(x \in U_1)L_{x,R} = L_{x,R'}) \Rightarrow R = R^{'}$ Esto quiere decir que si todas las etiquetas lingüísticas de las conexiones que conectan a una Regla R son iguales a las etiquetas de una Regla R' es porque $R = R'$.

7. Para todas las unidades de Regla $R \in U_2$ y todas las unidades $C, C^{'} \in U_3$ se cumple que: $(W(R, C) = 1) \wedge (W(R, C^{'}) = 1) \Rightarrow C = C^{'}$ Esto quiere decir que

   solo debe haber una conexión que salga de la unidad de Regla que conecte a una neurona de salida y que su peso sea de 1.

8. Para todas las unidades de salida $C \in U_3$, la activación $o_c = a_c = net_c$.

9. Para todas las unidades de salida $C \in U_3$, la entrada net_c es calculada como: $net_c = \frac{\sum\limits_{R \in U_2} W(R,C) * o_R}{\sum\limits_{R \in U_2} W(R,C)}$

10. La base de reglas puede ser definida por el usuario o de forma combinada usuario-preprocesamiento. Igualmente dado un conjunto de patrones de aprendizaje $\zeta = \{(p_1, t_1)(p_2, t_2), ..., (p_s, t_s)\}$ que consiste en un patrón de entrada $p \in \mathbb{R}^n$ y una respuesta esperada $t \in \{0, 1\}^m$ el algoritmo crea k unidades de regla en los siguientes pasos.

3.3. PREPROCESAMIENTO

Este paso tiene como objetivo crear una base de reglas, sobre las cuales se puede fundamentar para realizar el entrenamiento y afinación de los conjuntos fuzzy.

1. Seleccionar el siguiente patrón (p, t) de ζ, donde p representa el conjunto de datos de entrada, t representa la salida deseada de la red cuando se propague el respectivo conjunto de datos de entrada y ζ representa el conjunto de patrones de datos.

2. Para cada neurona de entrada $x_i \in U_1$ encontrar la función de pertenencia $\mu_{ji}^{(i)}$ tal que: $\mu_{ji}^{(i)}(p_i) = \underset{j \in \{1,...,q_i\}}{max} \{\mu_j^i(p_i)\}$ Es decir encontrar el conjunto que tenga mayor función de pertenencia cuando se propague el patrón.

3. Si el numero de reglas existentes es menor que k_{max} y no existe una unidad de regla con: $W(x_1, R) = \mu_{j1}^{(1)}, ..., W(x_n, R) = \mu_{jn}^{(n)}$ entonces se debe crear la unidad de regla y conectarlo a la unidad de salida c_l si $t_l = 1$, donde t_l es la salida esperada o clasificación del patrón propagado.

4. Si todavía hay patrones sin procesar en ζ, y $k < k_{max}$, entonces volver al paso (1), de lo contrario finalizar el proceso.

Determinar la base de reglas por medio de uno de los siguientes pasos:

Simple: consiste en dejar las k unidades de reglas creadas y parar cuando $k = k_{max}$.

Mejores Reglas: Procesar cada uno de los patrones ζ una vez más, y acumular la activación de cada una de las reglas para cada clase cuando se propaguen los patrones. Sí una unidad de regla R muestra una mayor activación acumulada para una clase C_j que para la clase C_R especificada por la conclusión de la regla, entonces se debe cambiar la conclusión de la regla R a C_j, esto quiere decir que se debe desconectar la regla R de la unidad de salida C_R y conectarla con la unidad de salida C_j. Procesar los patrones una vez más y calcular para cada unidad de regla:

$$V_R = \sum_{p \in \zeta} a_R^{(p)} e_p,\ e_p = \begin{cases} 1 & si\ es\ clasificado\ correctamente \\ 0 & otro\ caso \end{cases}$$

Mejores reglas por clase: Se procede de la misma forma que el método anterior, pero conservando para cada clase C_j las $\frac{k}{m}$ reglas, cuya conclusión representa la clase C_j.

3.4. Entrenamiento

El algoritmo de aprendizaje supervisado del sistema NEFCLASS-Q adapta los conjuntos fuzzy corriendo los datos de entrenamiento ζ cíclicamente hasta alcanzar algún criterio, por ejemplo que sobrepase el número máximo de errores, o que el error no pueda disminuir más a futuro.

Después de que el patrón es propagado a través de la red, el error es calculado por cada unidad de salida de forma:

$$\delta c_i = t_i - o_{ci}$$

donde o_{ci} es la activación de la neurona de salida i y δc_c el error de la neurona de salida i. Con base en este error, para cada unidad de regla activada se calcula el grado de cumplimiento de la siguiente forma:

$$\delta_R = o_R(1 - o_R) \sum_{c \in U_3} w(R, c)\delta c.$$

La función de pertenencia que es responsable del grado de cumplimiento es identificada con la siguiente condición:

$$W(x', R)(a_{x'}) = \min_{x \in U_1} \{W(x, R)(o_x)\}$$

es decir se busca el conjunto que tenga menor función de pertenencia con el patrón que se está propagando y solamente este conjunto es adaptado en función del error modificándolo con los siguientes valores:

$$\delta b = \sigma \cdot \delta R \cdot (c - a) \cdot sgn(o_{x'} - b)$$

$$\delta a = -\sigma \cdot \delta R \cdot (c - a) + \delta b$$

$$\delta c = \sigma \cdot \delta R \cdot (c - a) + \delta b$$

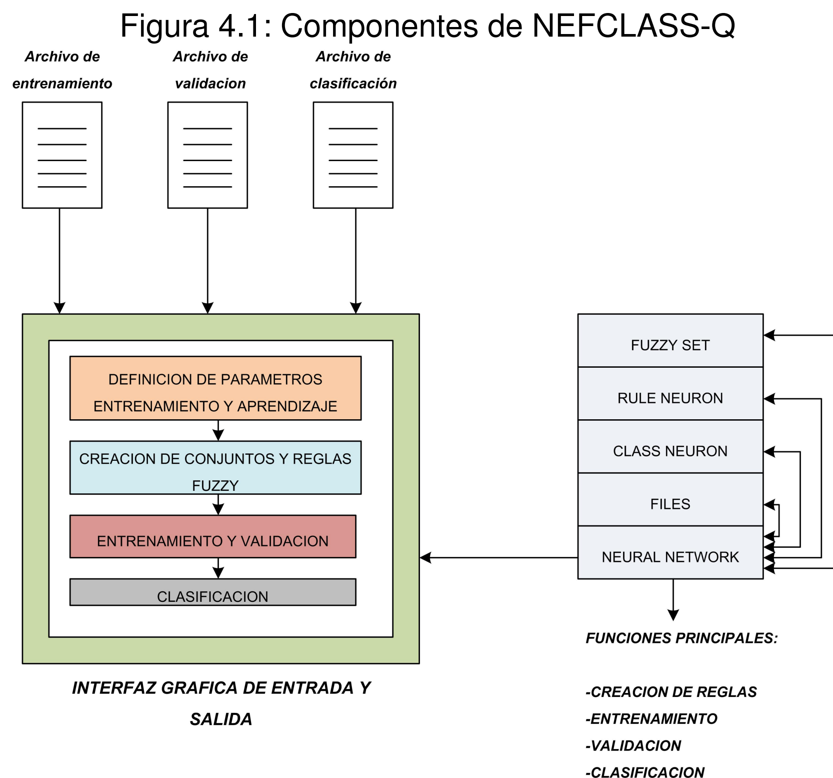
Un conjunto fuzzy es modificado solamente sí este no viola ninguna de las restricciones impuestas por el usuario, como lo podrían ser:

- Los conjuntos fuzzy no deben traslapar mas de cierto grado.
- Un conjunto fuzzy no debe sobrepasar el vecino.
- Los conjuntos fuzzy deben ser simétricos. etc.

4. NEFCLASS-Q

NEFCLASS-Q es la herramienta que se desarrolló en el presente proyecto, la cual tiene como objetivo principal la creación, entrenamiento y validación de una Red Neuro-Fuzzy para la clasificación de patrones de datos basada en el modelo NEFCLASS.[1] Esta herramienta está compuesta por cinco librerías las cuales engloban y sintetizan la arquitectura del perceptrón fuzzy y una interfaz gráfica que facilita la creación, aprendizaje y entrenamiento de la red por medio de un archivo de texto donde se encuentran los datos para el entrenamiento. Para el correcto funcionamiento del software, se deben disponer de 3 archivos como se puede ver en la figura 4.1. Con el fin de explicar cada uno de los procesos en detalle que realiza NEFCLASS-Q, se va a utilizar la base de datos IRIS.DAT como problema de clasificación.

Figura 4.1: Componentes de NEFCLASS-Q



Fuente: Autores

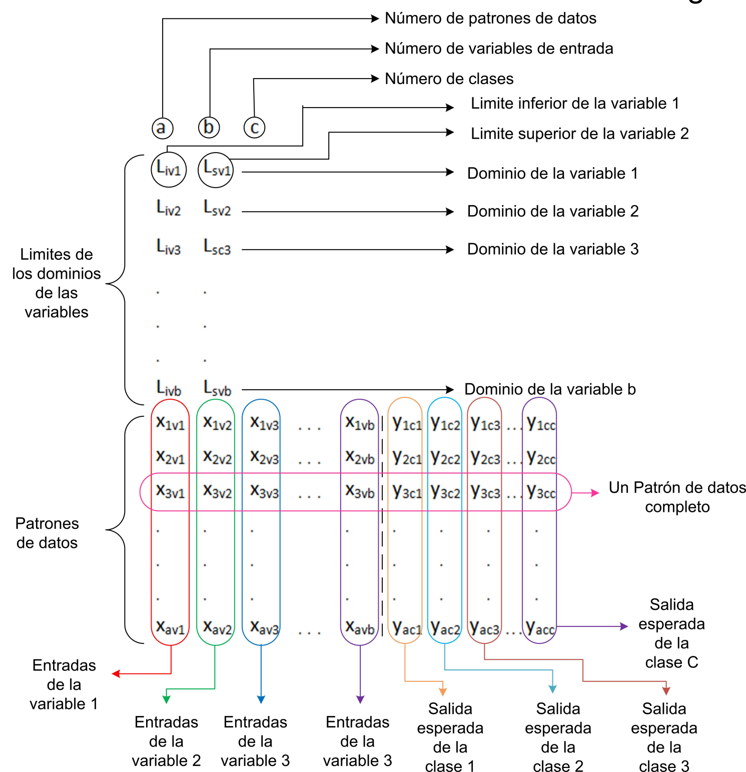
El archivo de entrenamiento es el que se encarga de proporcionar el conocimiento a la Red por medio de un proceso de ajuste de los pesos sinápticos[2] y debe tener

[1]NEFCLASS fue desarrollado por Rudolf Kruse y Detlef Nauck basándose en la arquitectura del perceptrón fuzzy simple elaborada por los mismos autores de NEFCLASS.

[2]Para la arquitectura del perceptrón fuzzy los pesos sinápticos son conjuntos fuzzy

la estructura de la figura 4.2. Esta figura muestra de forma general como debe ser el orden de los datos de entrenamiento para suministrarlos a NEFCLASS-Q mediante un archivo de texto. En la primera línea el valor ***a*** indica el número de patrones de datos que hay en el archivo, el valor ***b*** indica el número de variables de entrada del problema y el valor de ***c*** indica el número de clases del problema. En el siguiente grupo de líneas deben ir los dominios de cada una de las variables representados por L_{iv1} que significa límite inferior de la variable 1 y L_{sv1} que significa límite superior de la variable 1, el número de líneas de este grupo debe ser igual al valor ***b***. El siguiente grupo de líneas son conformadas por los patrones de datos, cada patrón de datos está dividido en dos partes, en la parte izquierda van las entradas y en la parte derecha van las salidas esperadas de cada patrón. La nomenclatura de las entradas es por ejemplo x_{1v1} significa entrada del patrón 1 a la variable 1 y x_{1v2} significa entrada del patrón 1 a la variable 2. Por otro lado en la sección de salidas y_{1c1} significa salida esperada del patrón 1 en la clase 1, estas salidas deben tener valores de 1 o 0, la sección de salida debe tener tantas columnas como diga el valor ***c***.

Figura 4.2: Estructura del archivo de entrenamiento genérica

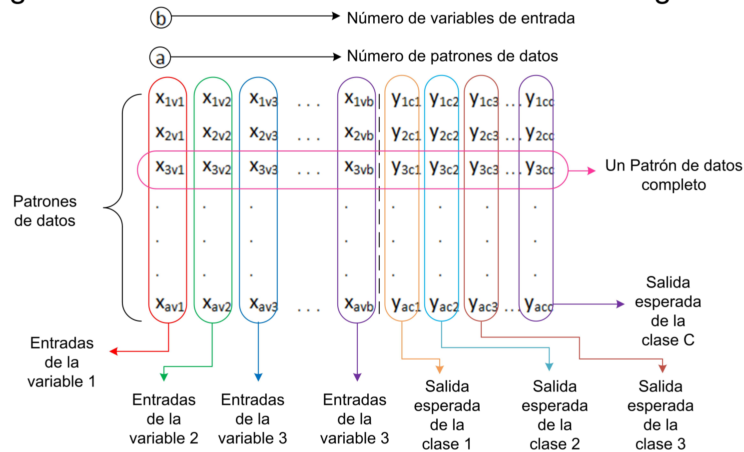


Fuente: Autores

El archivo de validación es el soporte para evaluar el entrenamiento realizado por NEFCLASS-Q. Este archivo es similar al de entrenamiento y tiene la estructura de la figura 4.3. En esta figura la primera línea contiene el valor ***b*** y determina el

número de variables de entrada, en la segunda línea contiene el valor ***a*** y determina el número de patrones. En las siguiente líneas se sitúan los patrones de datos que tiene el mismo orden que en el archivo de entrenamiento de la figura 4.2.

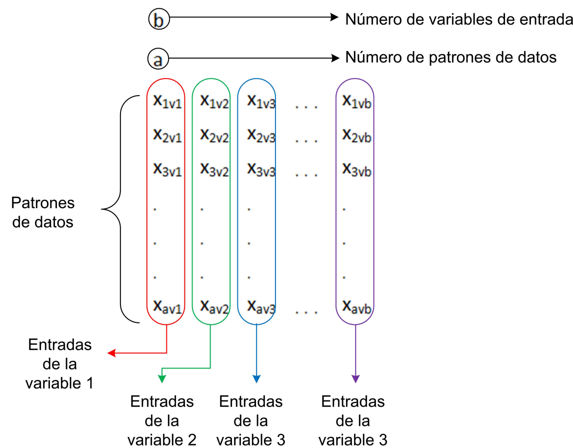
Figura 4.3: Estructura del archivo de validación genérica



Fuente: Autores

El último archivo es el de clasificación, en este archivo se ubican los patrones de datos que se quieren clasificar, este archivo tiene la estructura de la figura 4.4. Como podemos ver la única diferencia que tiene con respecto al archivo de validación de la figura 4.3 es que los patrones de datos no tienen clase esperada.

Figura 4.4: Estructura del archivo de clasificación genérico



Fuente: Autores

En el caso particular de IRIS.DAT una base de datos muy conocida en el campo de la clasificación de patrones de datos, ya que es muy utilizada para la evaluación de software de este tipo, que consiste en las medidas de algunas partes de tres tipos de flores.

- Iris Setosa
- Iris Versicolour
- Iris Virginica

Las cuales pueden ser clasificadas conociendo características como el ancho y la longitud del sépalo y del pétalo, en el archivo se encuentran los datos en el siguiente orden como se puede ver en la figura 4.5:

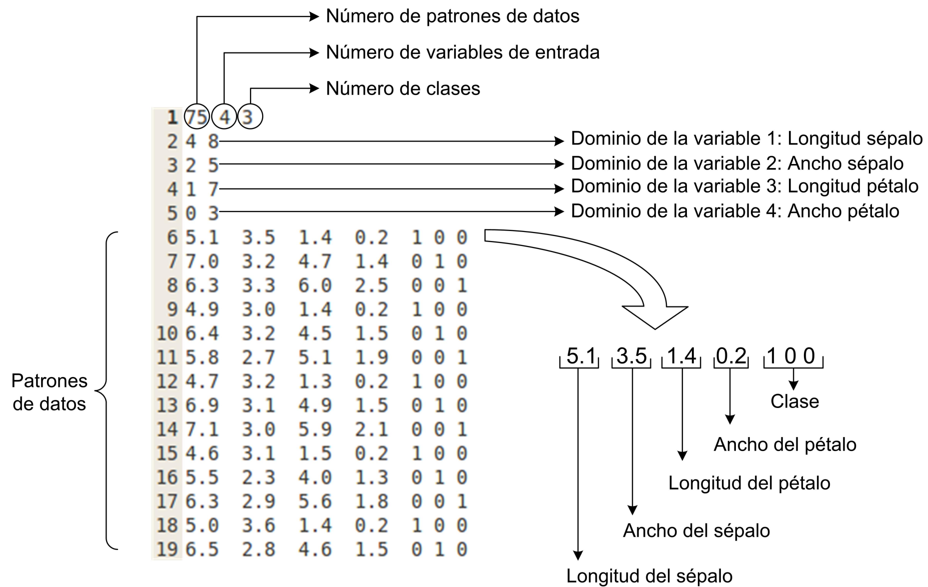
- Longitud del sépalo en cm.
- Ancho del sépalo en cm.
- Longitud del pétalo en cm.
- Ancho del pétalo en cm.

cada flor tiene codificada un vector de 3 números que identifica cada una. La codificación es la siguiente:

- Clase 1 0 0 es Iris Setosa
- Clase 0 1 0 es Iris Versicolour
- Clase 0 0 1 es Iris Virginica.

El archivo de entrenamiento para IRIS.DAT es el de la figura 4.5, en la línea 1, se encuentran el número de patrones que contiene el archivo (75), el número de variables (4) y el número de clases (3) respectivamente, de la línea 2 a la 5 se encuentran los dominios para cada una de las variables que en este caso son 4, donde la variable 1 representa la longitud del sépalo, la variable 2 representa el ancho del sépalo, la variable 3 representa la longitud del pétalo y la variable 4 representa el ancho del pétalo, de la línea 6 en adelante se encuentran los patrones de datos, que en este caso contiene 4 columnas, en cada columna están digitados los datos para cada variable y la columna 5 contiene un vector que define la clase a la que pertenece, por ejemplo para el patrón número 1 el vector que tiene es 1 0 0 que nos indica que el patrón pertenece a la clase 1 que indica Iris Setosa, la clase 2 indica Iris Versicolour y la clase 3 indica Iris Virginica.

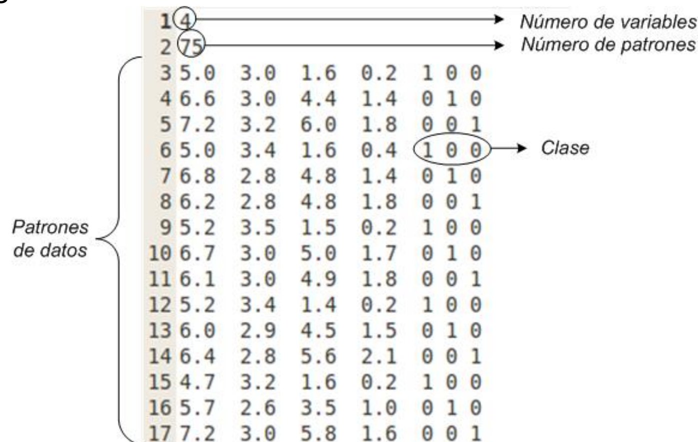
Figura 4.5: Estructura archivo de entrenamiento de IRIS



Fuente: Autores

El archivo de validación es el que se encarga de proporcionar los datos para evaluar el entrenamiento de la red, la estructura de este archivo es el de la figura 4.6. La primera línea contiene el número de variables, la segunda línea contiene el número de patrones de datos, de la línea 3 en adelante contiene los patrones de datos con las respectivas clases.

Figura 4.6: Estructura del archivo de validación de IRIS



Fuente: Autores

El archivo de clasificación es similar al de validación con la única diferencia que no se escriben a que clase pertenece cada patrón de datos, como se puede ver en la figura 4.7

Figura 4.7: Estructura del archivo de clasificación de IRIS

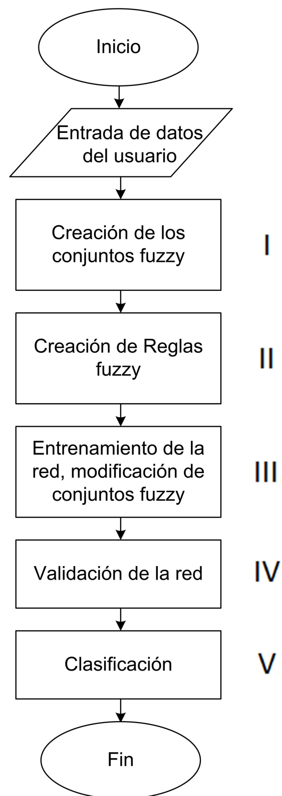
```
 1 4                     → Número de variables
 2 75                    → Número de patrones
 3 5.0  3.0  1.6  0.2
 4 6.6  3.0  4.4  1.4
 5 7.2  3.2  6.0  1.8
 6 5.0  3.4  1.6  0.4
 7 6.8  2.8  4.8  1.4
 8 6.2  2.8  4.8  1.8
 9 5.2  3.5  1.5  0.2
10 6.7  3.0  5.0  1.7
11 6.1  3.0  4.9  1.8
12 5.2  3.4  1.4  0.2
13 6.0  2.9  4.5  1.5
14 6.4  2.8  5.6  2.1
15 4.7  3.2  1.6  0.2
```

Patrones de datos (líneas 3–15)

Fuente: Autores

Con estos archivos podemos realizar los principales procesos de NEFCLASS como la creación de la red, creación de conjuntos y reglas fuzzy, entrenamiento de la red etc, como se pude ver en la figura 4.8.

Figura 4.8: Procesos de NEFCLASS-Q

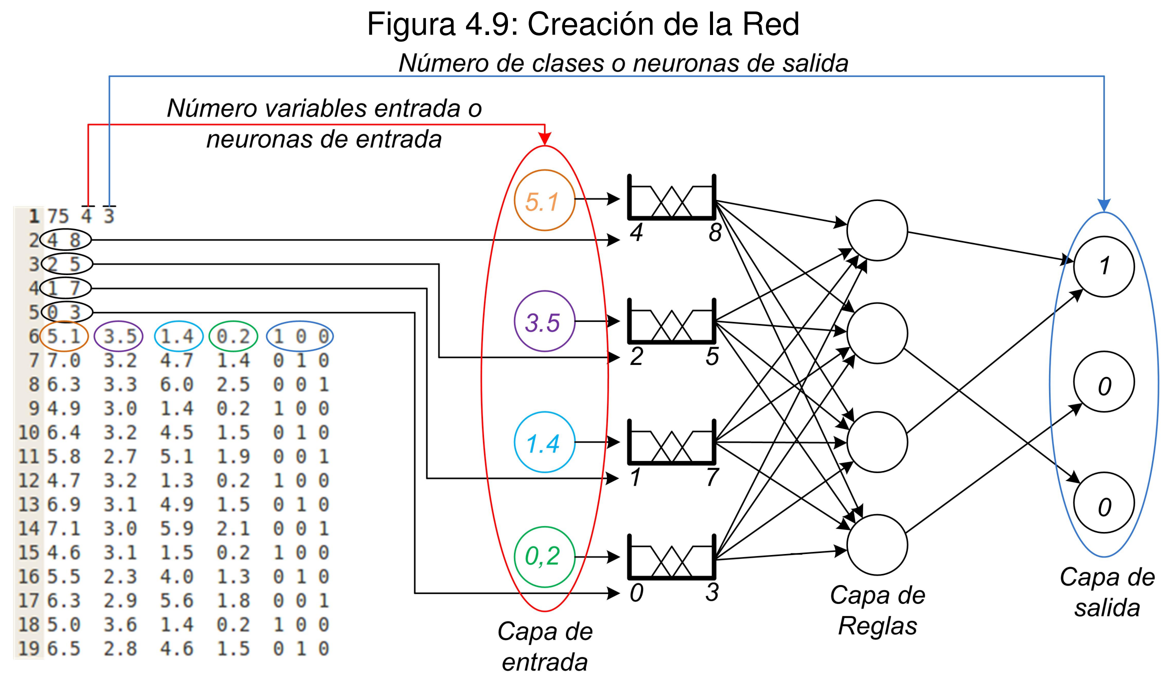


Fuente: Autores

4.1. CREACIÓN DE LA RED NEURO-FUZZY

Cuando un archivo de entrenamiento es cargado, automáticamente se definen el número de neuronas de entrada que representan el número de variables en el archivo, igualmente se definen el número de neuronas de salida que representan el número de clases en el archivo como se puede ver en la figura 4.9.

Figura 4.9: Creación de la Red

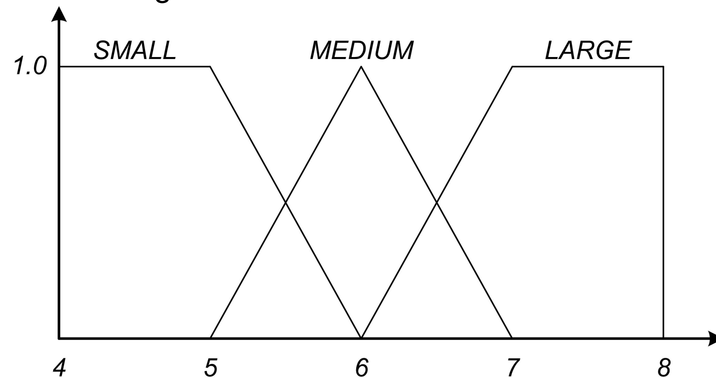


Fuente: Autores

4.1.1. Creación de los conjuntos fuzzy (I)

Los pesos sinápticos entre la capa de entrada y la capa oculta son representados por los conjuntos fuzzy, ellos permiten definir los valores que se están propagando en términos lingüísticos. El número de conjuntos fuzzy para cada unidad de entrada es definido por el usuario, y la forma de estos es definida inicialmente por NEFCLASS-Q de tal manera que queden equitativamente distribuidos en el dominio de cada variable que se encuentra en el archivo de entrenamiento. Por ejemplo vamos a tomar el archivo de entrenamiento de la figura 4.5, y definiremos los conjuntos de la variable 1 (Longitud sépalo), cuyo dominio se encuentra entre 4 y 8, y suponemos que el número de conjuntos definidos por el usuario fue de 3, teniendo esto NEFCLASS-Q puede representar los conjuntos fuzzy para la variable 1 como en la figura 4.10.

Figura 4.10: Variable 1 fuzzificada



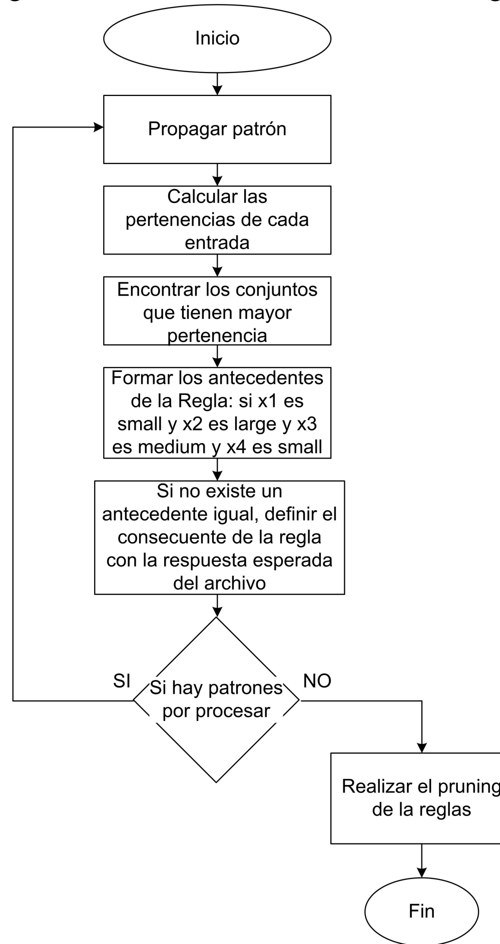
Fuente: Autores

4.1.2. Creación de Reglas(II)

Para definir el número de neuronas de la capa oculta que representan las Reglas Fuzzy[3], NEFCLASS-Q tiene un proceso de creación de reglas que esta explicada en la figura 4.11. El número de neuronas de la capa oculta va a ser igual al número de reglas creadas en el proceso o un valor máximo que es definido por el usuario. Este proceso es muy importante, ya que es en el que la Red Neuro-Fuzzy adquiere el conocimiento haciendo propagar los patrones de datos del archivo de entrenamiento, cuando NEFCLASS-Q propaga un patrón de datos, calcula las pertenencias de cada una de las entradas a sus respectivos conjuntos, cuando ha terminado de calcular todas las pertenencias, NEFCLASS-Q escogen un conjunto por cada variable, teniendo en cuenta que el conjunto tenga la mayor pertenencia, cuando se ha escogido los conjuntos, estos se utilizan para formar el antecedente de la regla, por último se verifica que este antecedente no exista para poder guardarlo en la base de conocimiento de la Red Neuro-Fuzzy, este proceso NEFCLASS-Q lo realiza cuando está propagando cada patrón de datos del archivo de entrenamiento.

[3]Para Identificar la capa oculta por favor remitirse a la sección 3.1 donde esta explicada la arquitectura de la Red Neuro.Fuzzy

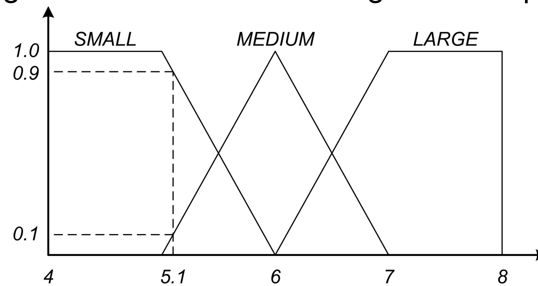
Figura 4.11: Proceso creación de reglas



Fuente: Autores

Para demostrar cómo NEFCLASS-Q crea una Regla, se va a propagar el patrón número uno, que se encuentra en la línea 6 de la figura 4.5. Los valores del patrón son 5.1, 3.5, 1.4, 0.2 que ingresan a las neuronas de entrada como se ve en la figura 4.9. El siguiente paso es calcular las pertenencias a los conjuntos de las variables, para la variable 1 el valor ingresado es 5.1, la pertenencia a *small* es 0.9, a *medium* es 0.1, a *large* es 0 como se puede ver en la figura4.12.

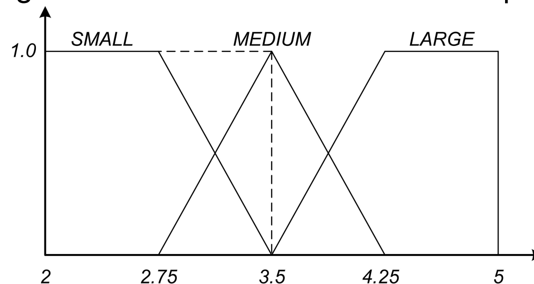
Figura 4.12: Variable 1. Longitud de sépalo



Fuente: Autores

Para la variable 2 el valor ingresado es 3.5, la pertenencia a *small* es 0, a *medium* es 1, a *large* es 0, como se puede ver en la figura4.13.

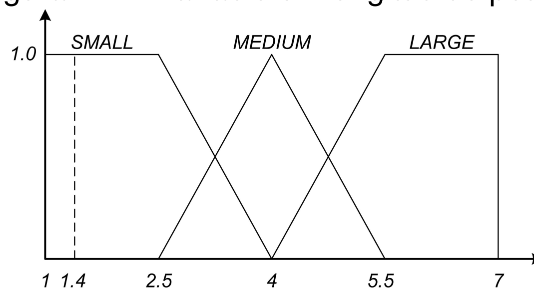
Figura 4.13: Variable 2. Ancho de sépalo



Fuente: Autores

Para la variable 3 el valor ingresado es 1.4 y la pertenencia a *small* es 1.0, a *medium* es 0 y a *large* es 0, como se puede ver en la figura 4.14.

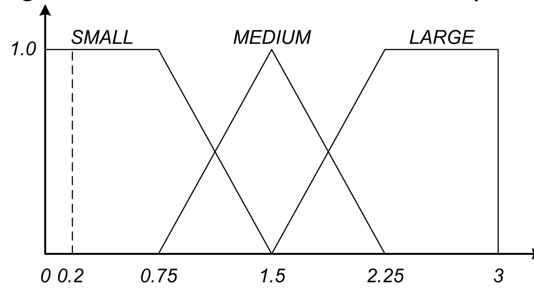
Figura 4.14: Variable 3. Longitud de pétalo



Fuente: Autores

Para la variable 4 el valor ingresado es 0.2 y la pertenencia a *small* es 1.0, a *medium* es 0 y a *large* es 0, como se puede ver en la figura 4.15.

Figura 4.15: Variable 4. Ancho de pétalo



Fuente: Autores

El siguiente paso que hace NEFCLASS-Q es escoger los conjuntos que tiene mayor pertenencia en cada dominio, para mejor entendimiento se presenta la tabla 4.1 que tiene las pertenencias a los conjuntos de todas las variables y presenta las mayores pertenencias de cada dominio en negrilla.

Tabla 4.1: Conjuntos que poseen mayor pertenencia

Pertenencias a los conjuntos				
Variable \ Etiqueta		small	medium	large
V1	Longitud sepalo	**0.9**	**0.1**	0
V2	Ancho de sepalo	0	**1**	0
V3	Longitud petalo	**1**	0	0
V4	Ancho petalo	**1**	0	0

Fuente: Autores

como se puede ver en la tabla 4.1, el conjunto que tiene mayor pertenencia para la variable 1 (Longitud del sépalo) es small, para la variable 2 (Ancho del sépalo) es medium, para la variable 3 (Longitud del pétalo) es small, para la variable 4 (Ancho del pétalo) es small. Con esto NEFCLASS-Q puede formar el antecedente de la regla que queda de la siguiente forma:

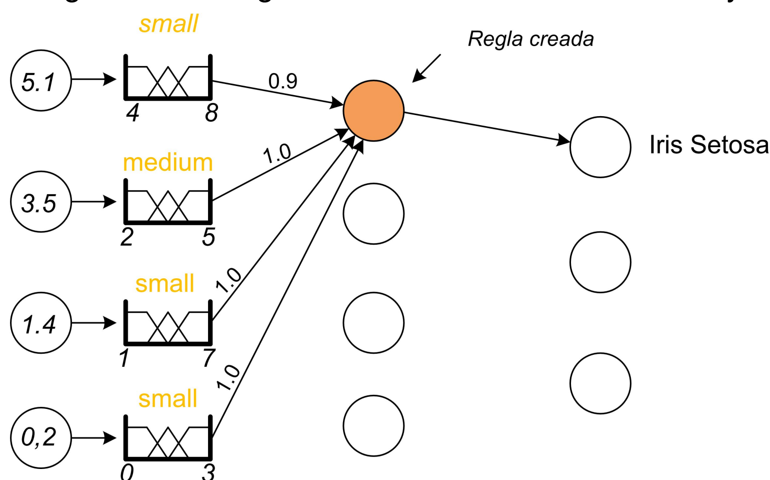
Si $longitud\ sépalo$ es $\mathbf{small}$ y $ancho\ del\ sépalo$ es $\mathbf{medium}$ y $longitud\ del\ pétalo$ es $\mathbf{small}$ y $ancho\ del\ pétalo$ es $\mathbf{small}$

Como esta regla es la primera que NEFCLASS-Q crea y no existe, entonces NEFCLASS-Q define el consecuente que está en la clase esperada del archivo de entrenamiento, en esta caso es clase 1 que indica Iris Setosa. La regla queda finalmente como:

Si $longitud\ sépalo$ es $\mathbf{small}$ y $ancho\ del\ sépalo$ es $\mathbf{medium}$ y $longitud\ del\ pétalo$ es $\mathbf{small}$ y $ancho\ del\ pétalo$ es $\mathbf{small}$ entonces $\mathbf{Iris\ Setosa}$

Esta regla puede ser representa en la red Neuro-Fuzzy como muestra la figura 4.16.

Figura 4.16: Regla creada en la Red Neuro-Fuzzy



Fuente: Autores

De esta forma NEFCLASS-Q crea una Regla Fuzzy, así mismo se propagan los demás patrones del archivo de entrenamiento y se crean todas las Reglas y queda definida completamente la base de conocimiento de la Red Neuro_Fuzzy.

Con la finalidad de que el conocimiento sea más concreto para el usuario, NEFCLASS-Q hace un proceso de pruning de reglas, en el cual, puede dejar una base de reglas que contenga las reglas más importantes, para esto hay 3 métodos:

- El método de *Todas las Reglas* o *Simple* que deja la base de conocimiento tal como se creó, con el mismo número de reglas.
- El método de las *Mejores Reglas* que deja determinado número de reglas a pedido del usuario, y NEFCLASS.Q escoge las reglas que más han tenido activación cuando se propagan los patrones.
- El método de la *Mejores Reglas por clase* que limita a dejar las reglas que hayan tenido más activación para cada clase, donde el número de reglas por clase es definido por el usuario.

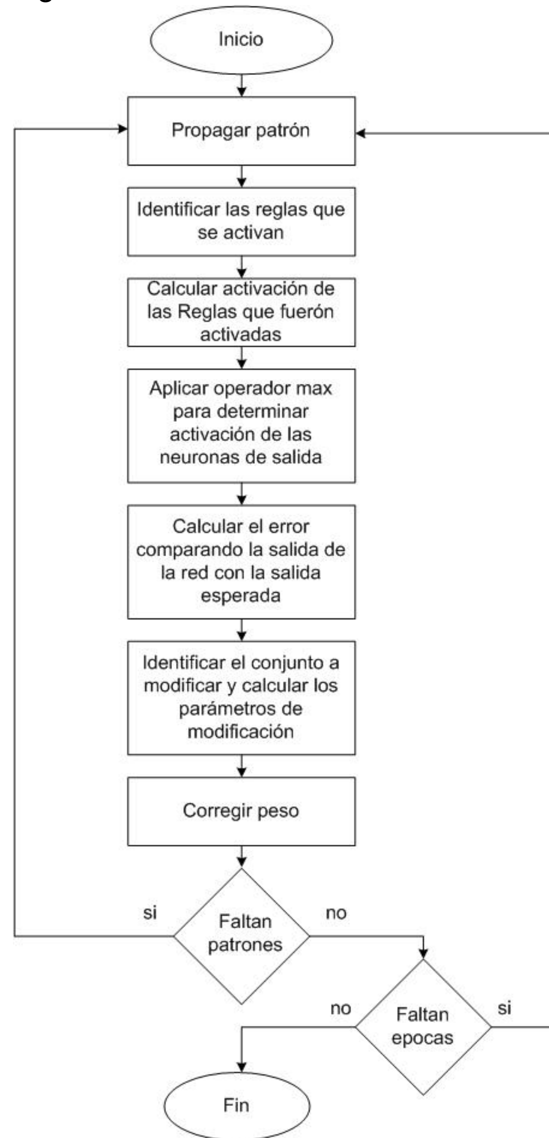
Con el pruning de Reglas, se finaliza el proceso se creación de Reglas según el diagrama de flujo de la figura 4.11.

4.1.3. Entrenamiento de la red(III)

Teniendo la base del conocimiento definida, NEFCLASS-Q prosigue a realizar el entrenamiento de la red, en el que se hacen propagar los patrones de datos del archivo de entrenamiento para que la red proporcione una respuesta, esta es comparada con la respuesta deseada que se encuentra en el archivo de entrenamiento para generar un error, este error es retropropagado para modificar los pesos de a

cuerdo a las formulas planteadas en el capítulo 3, este proceso es explicado en la figura 4.17.

Figura 4.17: Proceso de entrenamiento



Fuente: Autores

Para demostrar cómo NEFCLASS-Q realiza el entrenamiento de la red Neuro-Fuzzy vamos a suponer una base de conocimiento como el de la tabla 4.2, con 4 reglas y con los mismos conjuntos fuzzy mostrados en las figuras 4.12, 4.13, 4.14, 4.15.

Tabla 4.2: Base de conocimiento

	Longitud sepalo	Ancho sepalo	Longitud petalo	Ancho petalo	Consecuente
1	small	medium	small	small	Iris setosa
2	large	medium	medium	medium	Iris Versicolour
3	medium	medium	large	large	Iris Virginica
4	small	small	small	small	Iris Setosa

Fuente: Autores

NEFCLASS-Q prosigue a propagar nuevamente el patrón número uno del archivo de entrenamiento para calcular cada una de las pertenencias a los conjuntos, para mayor entendimiento se presenta la figura 4.18, que contiene las pertenencias calculadas a los conjuntos y como se puede ver en la figura 4.18, las pertenencias que están resaltadas en negrilla son las que tiene un valor mayor a cero, y van a jugar un papel importante en este proceso de entrenamiento.

Figura 4.18: Conjuntos que tiene pertenencia mayor a cero

Pertenencias a los conjuntos				
Variable / Etiqueta		small	medium	large
V1	Longitud sepalo	0.9	0.1	0
V2	Ancho de sepalo	0	1	0
V3	Longitud petalo	1	0	0
V4	Ancho petalo	1	0	0

Fuente: Autores

- Para la variable: 1 Longitud del sépalo se activan los conjuntos small y medium.
- Para la variable: 2 Ancho del sépalo se activan el conjunto medium.
- Para la variable: 3 Longitud del pétalo se activa el conjunto small.
- Para la variable: 4 Ancho del pétalo se activa el conjunto small.

Basándose en la tabla 4.18 se tienen dos posible antecedentes de reglas. La primera regla (Regla A) se puede generar con los valores resaltados de la tabla 4.19.

Figura 4.19: Formación de la regla A

Pertenencias a los conjuntos				
Variable / Etiqueta		small	medium	large
V1	Longitud sepalo	0.9	0.1	0
V2	Ancho de sepalo	0	1	0
V3	Longitud petalo	1	0	0
V4	Ancho petalo	1	0	0

Regla A

Fuente: Autores

Por lo tanto el antecedente de la regla A quedara de la siguiente forma:

Si $longitud\ sépalo$ es $\mathbf{small}$ y $ancho\ del\ sépalo$ es $\mathbf{medium}$ y $longitud\ del\ pétalo$ es $\mathbf{small}$ y $ancho\ del\ pétalo$ es $\mathbf{small}$

La segunda regla (Regla B) se puede generar con los conjuntos que tienen las pertenencias resaltadas de la tabla 4.20.

Figura 4.20: Formación de la regla B

Pertenencias a los conjuntos				
Variable / Etiqueta		small	medium	large
V1	Longitud sepalo	0.9	0.1	0
V2	Ancho de sepalo	0	1	0
V3	Longitud petalo	1	0	0
V4	Ancho petalo	1	0	0

Regla B

Fuente: Autores

El antecedente de la regla B quedara de la siguiente forma:

Si $longitud\ sépalo$ es $\mathbf{medium}$ y $ancho\ del\ sépalo$ es $\mathbf{medium}$ y $longitud\ del\ pétalo$ es $\mathbf{small}$ y $ancho\ del\ pétalo$ es $\mathbf{small}$

Teniendo los antecedentes de las Reglas A y B de la tabla 4.3 como las posibles Reglas que se van a activar, debemos compararlas con la base de conocimiento de la tabla 4.2, en el caso de que las Reglas A y B estén en la base de conocimiento se podrá decir que estas Reglas se activarán.

Tabla 4.3: Posibles antecedentes de reglas

	Longitud sepalo	Ancho sepalo	Longitud petalo	Ancho petalo
A	small	medium	small	small
B	medium	medium	small	small

Fuente: Autores

Comparando la tabla 4.3 con la tabla 4.2 se puede ver que el antecedente de la Regla A de la tabla 4.3 es igual al antecedente de la Regla 1 de la tabla 4.2, pero el antecedente de la Regla B de la tabla 4.3 no es igual a ningún antecedente de la base de conocimiento de la tabla 4.2. Por lo tanto se puede decir que la Regla 1 de la tabla 4.2 se activará.

Para calcular la activación de la regla 1 NEFCLASS-Q escoge el valor más pequeño de las pertenencias a los conjuntos que componen la regla 1, para mayor entendimiento se presenta la tabla 4.4, esta tabla se construyó a partir de los valores resaltados de las pertenencias de la Regla en la figura 4.19, que fue la que se activó.

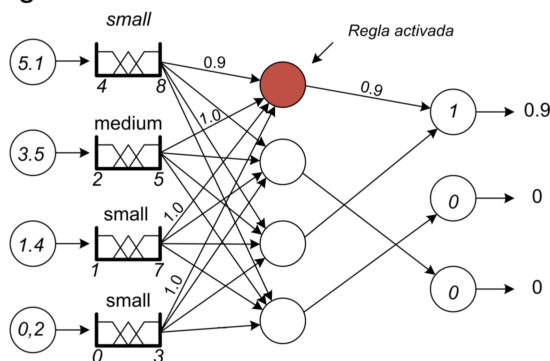
Tabla 4.4: Activaciones de los conjuntos que componen la regla

	Longitud sepalo	Ancho sepalo	Longitud petalo	Ancho petalo
Conjunto	small	medium	small	small
Pertenencia	0.9	1	1	1

Fuente: Autores

como se puede ver en la tabla 4.4, el valor más pequeño es de 0.9 que se debe a la pertenencia al conjunto *small* de la variable *Longitud del sépalo,* por ende la activación de la regla será 0.9 . Como en esta caso solo se activó una regla y considerando la aplicación del operador fuzzy *max* la salida de la red será 0.9 0 0 como se puede ver en la figura 4.21

Figura 4.21: Salida de la red neuro-fuzzy



Fuente: Autores

como la salida esperada es 1 0 0 y debido a que son 3 neuronas de salida se calcula 3 errores con la formula $\delta c_i = t_i - o_{ci}$.

Para la neurona de salida 1 el error es: $\delta c_1 = 1 - 0{,}9 = 0{,}1$

Para la neurona de salida 2 el error es: $\delta c_2 = 0 - 0 = 0$

Para la neurona de salida 3 el error es: $\delta c_3 = 0 - 0 = 0$.

A continuación NEFCLASS-Q debe calcular el grado de cumplimiento para la regla activada con la formula.

$$\delta_R = o_R(1 - o_R)\delta c$$

donde o_R es la activación de la regla calculada anteriormente como 0.9 y δc es el error en la salida del consecuente de la regla, es decir en este caso la Regla 1 tiene como consecuente la clase 1 (Iris Setosa), entonces el error que se debe utilizar es δc_1, que fue calculado anteriormente con un valor de 0.1, entonces el grado de cumplimiento de la Regla 1 se calcula como:

$$\delta_R = 0{,}9(1 - 0{,}9)0{,}1 = 0{,}009$$

Ahora NEFCLASS-Q pasa a escoger el conjunto fuzzy que se va a modificar y a calcular los parámetros de modificación, en este caso es el conjunto small de la variable 1, ya que es el que tiene menor pertenencia como se puede ver en la tabla 4.4, por otro lado el cálculo de los parámetros de modificaciones se hace suponiendo que el usuario ha definido un coeficiente de aprendizaje de 0.05. Para calcular los parámetros de modificación NEFCLASS-Q utiliza las siguientes formulas:

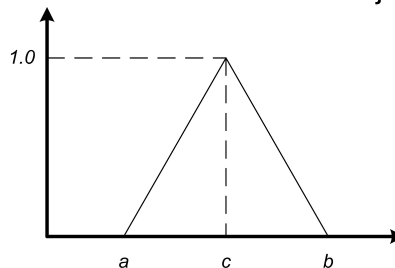
$$\delta b = \sigma \cdot \delta R \cdot (c - a) \cdot sgn(o_{x'} - b)$$

$$\delta a = -\sigma \cdot \delta R \cdot (c - a) + \delta b$$

$$\delta c = \sigma \cdot \delta R \cdot (c - a) + \delta b$$

donde los valores δa, δb, δc son las modificaciones que se la va a realizar al conjunto, σ representa el coeficiente de aprendizaje definido por el usuario, a, b, c son los parámetros del conjunto fuzzy y están definidos en la figura 4.22.

Figura 4.22: Parámetros de un conjunto fuzzy

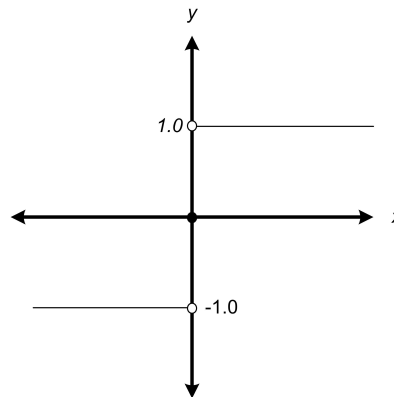


Fuente: Autores

$o_{x'}$ representa el valor del patrón de entrada a la variable que contiene el conjunto a modificar que en este caso es 5.1, la función $sgn(x)$ es la función signo y esta definida como:

$$f(x) = \begin{cases} 1 \; si \; x > 0 \\ 0 \; si \; x = 0 \\ -1 \; si \; x < 0 \end{cases}$$

Figura 4.23: Función signo



Fuente: Autores

y por último el valor δ_R es el grado de cumplimiento de la regla calculada anteriormente como 0.009, así los valores son calculados como:

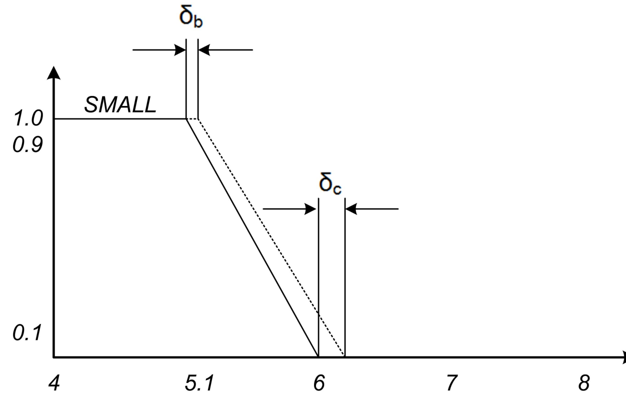
$$\delta b = \sigma \cdot \delta R \cdot (c - a) \cdot sgn(o_{x'} - b) = 0{,}05 \cdot 0{,}009 \cdot (6 - 4) \cdot sgn(5{,}1 - 5) = 0{,}0009$$

$$\delta a = -\sigma \cdot \delta R \cdot (c - a) + \delta b = -0{,}05 \cdot 0{,}009 \cdot (6 - 4) + \delta b = 0$$

$$\delta c = \sigma \cdot \delta R \cdot (c - a) + \delta b = 0{,}05 \cdot 0{,}009 \cdot (6 - 4) + \delta b = 0{,}0018$$

Con estos valores NEFCLASS-Q puede modificar el conjunto y queda finalmente como el de la figura 4.24.

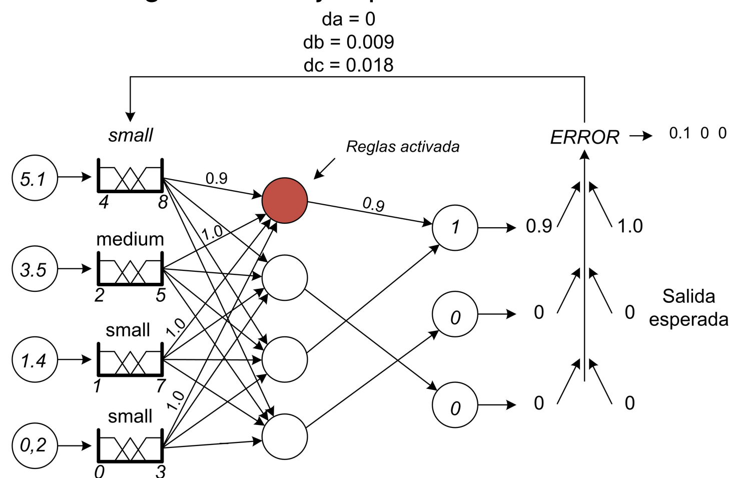
Figura 4.24: Conjunto modificado por entrenamiento



Fuente: Autores

Un resumen de la propagación de un patrón en el entrenamiento se encuentra en la figura 4.25

Figura 4.25: Ejemplo entrenamiento



Fuente: Autores

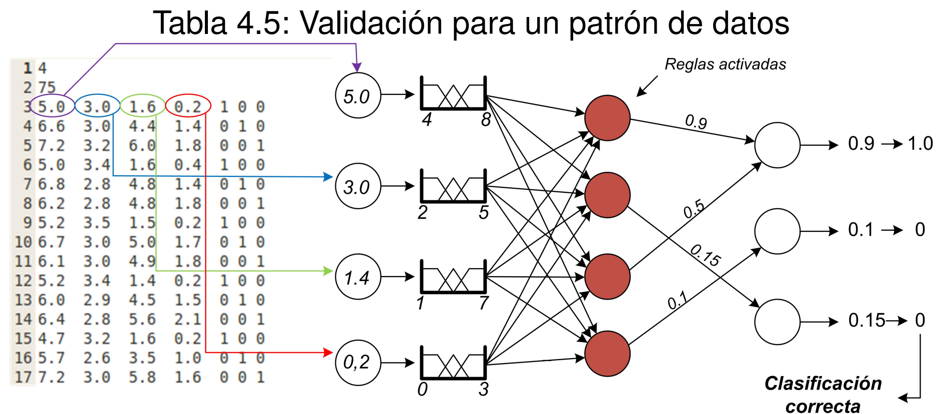
Después de que NEFCLASS-Q propague todos los patrones el número de veces que diga el número de épocas, se finaliza el proceso de entrenamiento como muestra la figura 4.17. Después de que NEFCLASS-Q ha entrenado la red se procede a realizar la validación del entrenamiento, donde se van a obtener resultados para que el usuario decida si se necesita entrenar más la red o si pasa al proceso de clasificación.

4.1.4. Validación del entrenamiento (IV)

Cuando NEFCLASS-Q ha terminado de entrenar la red, sigue el proceso de validación, en este proceso se utiliza el archivo de validación, donde se propagaran los

patrones de datos contenidos en este archivo de la misma forma como fue explicado en el proceso de entrenamiento con el fin de que la red de una respuesta, esta es comparada con la respuesta esperada contenida en el archivo de validación para saber si la red ha clasificado el patrón correctamente, después de haber propagado todos los patrones del archivo de validación, se calcula el porcentaje de los patrones clasificados correctamente. En la figura 4.5 se ilustra la validación para un patrón.

Tabla 4.5: Validación para un patrón de datos



Fuente: Autores

4.1.5. Clasificación (V)

En el proceso de clasificación es similar al de validación, donde los patrones son propagados y la red da una respuesta para cada uno, esta respuesta es la clasificación del patrón.

4.2. DOCUMENTACIÓN DEL CÓDIGO

Las librerías de NEFCLASS-Q están divididas en cinco módulos básicos: *NeuralNetwork, FuzzySet, Universe, Files, RuleNeuron*. Las cuales engloban y sintetizan la arquitectura del Perceptrón difuso, también contienen un ejemplo de aplicación de estas en el problema de la clasificación de patrones de la base de datos IRIS DATA, Estas librerías permiten crear modelos de sistemas híbridos Neuro-Fuzzy basadas en el algoritmo NEFCLASS.

Para la creación de los pesos, que realmente son conjuntos fuzzy, se utiliza la clase *FuzzySet,* donde además de representar los conjuntos fuzzy o pesos sinápticos, también nos va a permitir almacenar la etiqueta lingüística del conjunto. Esta clase tiene las siguientes atributos:

- La etiqueta lingüística.
- Los valores *a, b, c* que determinan la forma y dimensión del conjunto.

- La característica de si el conjunto esta en el extremo derecho, el extremo izquierdo o es un conjunto central.

Para la creación de un *objeto* en el Lenguaje de programación C++ se utilizan los constructores, estos nos permiten inicializar por defecto, o con valores requeridos para cada atributo del *objeto*. Para este caso particular vamos a inicializar los atributos o variables privadas de un *objeto FuzzySet* por medio de los siguientes constructores:

```
/* Constructores */
    FuzzySet();
    FuzzySet(std::string label);
    FuzzySet(double min, double mid, double max);
    FuzzySet(std::string label,double min, double mid, double max);
    FuzzySet(std::string label,
        double min, double mid, double max,FuzzyPosition position);
    FuzzySet(std::string label,
        bool isMinSet, double mid, double max, double LowLimit);
    FuzzySet(std::string label,
        double min, double mid, bool isMaxSet, double HigLimit);
    FuzzySet(const FuzzySet&);
```

Del mismo modo esta librería esta equipada con *funciones* o *métodos* que nos permiten modificar, obtener y calcular los atributos del *objeto,* por ejemplo las funciones *tuneA, tuneB, tuneC* funcionan como un afinador de conjuntos donde el parámetro que recibe es el valor diferencial a modificar, estos métodos son muy importantes en el proceso de entrenamiento de la red híbrida, ya que se debe modificar los atributos de cada conjunto para que la respuesta se aproxime cada vez más a la esperada. Dentro de los *métodos* para *FuzzySet* tenemos los siguientes:

```

/* Funcion de pertenencia */
    double membership(double x);

/* Funciones para modificar variables privadas */
    void setLabel(std::string label);
    void setValues(double A, double B, double C);
    void setValues(std::string label,double A, double B, double C);
    void setLeftShouldered();
    void setRightShouldered();
    void setMiddleSet();

/* Funciones que modifican los valores de 'a', 'b' o 'c'
   imponiendo una única restricción de forma del conjunto */
    bool tuneA(double da);
```

```
    bool tuneB(double db);
    bool tuneC(double dc);
    bool tune(double da, double db, double dc);

/* Funciones para obtener el valor de variable privadas */
    const char *getLabel();
    double getA();
    double getB();
    double getC();

    bool isLeftSet();
    bool isRightSet();
    bool isMiddleSet();
    FuzzyPosition getPosition();

/* Sobre carga de operadores de flujo, comparación y asignación */
    friend std::ostream& operator<< (std::ostream&, FuzzySet&);
    friend bool operator== (const FuzzySet&, const FuzzySet&);
    FuzzySet& operator =(const FuzzySet&);
```

Para definir el número de conjuntos que van a representar una determinada variable, el rango o universo de discurso, el máximo traslape que se debe tener entre conjuntos, entre otras características, utilizamos una clase llamada *Universe,* esta clase tiene los siguientes atributos:

- *restrictedOverlap* es un parámetro que permite activar o desactivar las restricciones de traslape.
- *maxOverlap* es el máximo valor de traslape que se permite cuando se estén modificando los conjuntos fuzzy.
- *NumFuzzySets* es el número de conjuntos fuzzy que componen el universo y definen la variable en términos lingüísticos.
- *LowLimit* es el limite inferior del universo de discurso.
- *HighLimit* es el limite superior del universo de discurso.
- *Fsets* son los Conjuntos Fuzzy definidos por la clase *FuzzySet* almacenados en un puntero a *FuzzySet.*

Para la creación de un objeto Universo nos ayudamos por medio de los constructores que son los siguientes:

```
/* Costructores */
    Universe();
```

```
    Universe(int numSets,double lowValue,double highValue);
    Universe(const Universe&);
```

Y a su vez tenemos las funciones o métodos que nos permiten obtener y definir variables privadas, afinar o modificar los conjuntos, entre otras. Los métodos de la clase *Universo* son las siguientes:

```
/* Fuciones para modificar variables privadas */
    void createFuzzySets(int numSets,double lowValue,double
       highValue);
    void setLowLimit(double lowValue);
    void setHighLimit(double highValue);
    void setLimits(double lowValue, double highValue);

    void setLabels(std::string *labels);
    void setRestrictedOverlap(double MaxOver);
    void setUnrestrictedOverLap();

    bool setFuzzyLabel(int numSet, std::string label);
    FuzzySet *fuzzySets();

/* Funciones para obtener el valor de variables privadas */
    double *setInput(double Input);
    void tuneSet(int numSet, double da, double db, double dc);
    double getMaxOverlap();

    int getNumFuzzySets();
    double getLowLimit();
    double getHighLimit();

/* Sobrecarga del operador de flujo */
    friend std::ostream& operator <<(std::ostream&, const Universe
       &);
    Universe& operator =(const Universe&);
```

Para construir o crear una regla completamente se debe tener definido los siguientes elementos:

- *Activación* es la activación de las reglas para una serie de entradas dadas la forma de la arquitectura y del patrón que se propague a través de la red.
- *AAC* es la activación acumulada por clase, que sera utilizado cuando se haya activado la opción “Best per class”, .

- AA es la activación acumulada total, es la sumatoria de todas las activaciones o respuestas de la *RuleNeuron*, este valor es utilizado cuando este activada la opción de la mejores reglas o “Best Rules”.

- *Antecedents* es el Antecedentes de la Regla y esta formado por una serie de números donde cada uno de ellos representa el conjunto activado por cada neurona de entrada.

- *Consecuent* es el consecuente que es un número entero que indica la clase a la que esta asociada la RuleNeuron.

- *InputNeurons* es el número de neuronas de entrada conectadas a la regla, este valor es indispensable a la hora de reservar memoria dinámica para el *Antecedents.*

- *Clases* es el número de clases posible que el patrón puede llegar a ser clasificado.

Para crear un *Ruleneuron* tenemos los siguientes constructores.

```
/* Constructores */
    RuleNeuron();
    RuleNeuron(int NumOfInputNeurons);
    RuleNeuron(int NumOfInputNeurons, int NumOfClassNeurons);
    RuleNeuron(const RuleNeuron &);
```

Las funciones y métodos de la clase *Ruleneuron* permitirán definir el número de items del antecedente, el número de posibles clasificaciones, la posibilidad de editar los antecedentes, obtener el valor de activación de la *RuleNeuron*, entre otras. Estas funciones están definidas de la siguiente forma:

```
/* Fuciones de parametros*/
    void setNumInputsNeurons(int NumOfInputNeurons);
    void setNumClasses(int NumClasses);
    bool setAntecedent(int NumOfInput, int NumOfSetActivated);
    void setConsecuent(int Class);
    void setClassesConsecuent(int NumClasses, int Class);

    void setInput(double Input);
    void setInput(double *Input);
    void resete();
    void reseteActivation();

/* Fuciones de obtencion */
    int getBestClass();
```

```
    double getActivation();
    double getActivation(double Input);
    double *getAcumulatedActivationPerClass();
    double getAcumulatedActivationClass(int Class);

    int* getAntecedents();

    double getAcumulatedActivation();

/* Funciones de obtencion de variables privadas */

    int getConsecuent();
    int getNumInputs();
    int getNumClasses();

/* Sobrecarga de operadores de flujo, comparacion y asignacion */
    friend std::ostream& operator <<(std::ostream&, const
        RuleNeuron&);
    friend bool operator ==(const RuleNeuron &, const RuleNeuron &)
        ;
    RuleNeuron& operator =(const RuleNeuron &);
```

La principal característica que tiene un *objeto ClassNeuron* es la activación que va a ser 1 o 0, las activaciones de las *ClassNeuron* determinarán la clasificación del patrón propagado, por ejemplo las salidas pueden ser 1 0 0, lo que indica que es clase 1. Para la creación de una *ClassNeuron*, se utilizan los siguientes constructores:

```
    ClassNeuron();
    virtual ~ClassNeuron();
```

Las funciones de esta clase ayudan a suministrar la entrada a la red y a obtener los valores de activación. Las funciones son las siguientes:

```
    void setInput(double Input);
    void resete();
    double getActivation();
    double getActivation(double Input);
```

Para la creación de una red Neuro-Fuzzy es necesario definir los parámetros mencionados anteriormente, NEFCLASS-Q contiene una librería llamada *Network,* esta librería tiene los siguientes atributos:

- *Epoch* es el número de epocas a las que va a ser sometido la red en el proceso de entrenamiento.
- *EpochStep* es el intervalo de épocas en el que se hará un realimentación de los datos enviados a un visor de cambios.
- *StepByStep* es un boleano que de ser verdadero activa la retroalimentación paso a paso, definido por la variable *EpochStep*.
- *INeurons* es el número de neuronas de la capa de entrada de la red, también representa el número de variables.
- *RNeurons* es el número de Reglas contenidas en la capa escondida.
- *CNeurons* es el número de clases por reconocer en la red, define el número de neuronas a de la capa de salida.
- *NumBestRules* es el número máximo de reglas que contendrá la red después del proceso de "truncado de reglas", la connotación de esta variable cambia de acuerdo al tipo de *pruning* que sea usado para definir el número final de reglas. Para mas información revisar la teoría referente al modelo.
- *RN* es un puntero a *RuleNeuron* encargado de almacenar el total de reglas del modelo.
- *U* Puntero a *Universe* encargado de almacenar los universos asociados a cada variable del sistema.
- *CN* es un puntero a *ClassNeuron* encargado de almacenar las neuronas de la capa de salida.
- *RulesPruning* es el tipo de truncado de reglas usado en el sistema, puede contener uno de los tres valores posibles, *AllRules*, *BestRules* y *BestRulesPerClass*.

Para crear una red, es necesario primero inicializar las variables, esto lo podemos hacer por medio de los constructores que tenemos disponibles en la librería *Network*.

```
/* Constructores */

Network();
Network(int numInputs);
Network(int numInputs, int numClasses);
Network(int numInputs, int numClasses, int maxRules, Pruning P)
    ;
Network(Files *handler, int maxRules, Pruning P);
Network(Files *handler, int maxRules, Pruning P, int maxEpoch);
```

La librería esta equipada con funciones que permiten obtener y modificar los atributos, crear la base de reglas, entrenamiento de la red etc.

```
/* Funciones de Utilidad */
void setInputNeurons(int numInputs);
void setClassNeurons(int classNeuron);
void setRuleNeurons(int maxRules, Pruning P);
void initialize();
void createRules(Files *handler);
void setLimits(double* lowValues, double* highValues);
void setNumEpoches(int maxEpoches);
void setSigma(double sigma);
void setSigmas(double sgA, double sgB, double sgC);
void addVariable(const Universe& Un);
void addRule(const RuleNeuron& R);
bool existRule(const RuleNeuron& R);
void activateRules(double** MP);
void setPruning(Pruning P, int maxRules = 0);
void inputPatterns(Files *handler);
void propagatePattern(double *pattern, int classPattern);
void pruning(Files *handler);
void training(Files *handler);
void training(Files *handler, int maxEpoches);
bool validate(double *pattern, int classPattern);
int getClass();
void aproximate();
double* calculateDeltaClass(int* target);
int* getMaxMembershipSets(double** MP);
int getMinMembershipSets(double** MP, RuleNeuron R);
int getMinMembershipSets(double* Pattern, RuleNeuron R);
double** getMP(double* inpt);
void reseteNeurons();
void reseteUniverses();
void resete();
void clearMatrix(double** M);
int sgn(double Value);
```

4.3. FUNCIONES DE NEFCLASS-Q

En este capitulo se explican las librerías creadas en este proyecto, y que fueron utilizadas en construcción de NEFCLASS-Q, esto con el fin de que los estudiantes de Ingeniería Mecánica de la Universidad Industrial de Santander que estén interesados en esta aplicación, puedan entender fácilmente el código, modificarlo y mejorarlo, con fines académicos.

Las librerías principales del NEFCLASS-Q son:

1. *Fuzzy Set*
2. *Universe*
3. *RuleNeuron*
4. *ClassNeuron*
5. *NeuralNetwork*
6. *Files*

4.4. LIBRERÍA *FUZZYSET*

Esta librería contiene los métodos y características para manejar conjuntos *Fuzzy* modelo Neuro-Fuzzy NEFCLASS-Q estos conjuntos tienen forma triangular, con tres posibles posiciones dentro del universo de discurso, esta posición determina la forma para calcular la pertenencia dentro del conjunto. La máxima pertenencia es 1 y es alcanzada cuando el valor que ingresa al conjunto es igual a la variable 'b' del conjunto. Para realizar las modificaciones de los parámetros la única restricción existente es de 'forma', es decir que la variación de uno de sus atributos no altere la forma triangular o trapezoidal.

4.4.1. Constructores *FuzzySet*

Para la creación de un conjunto *Fuzzy* utilizamos los constructores por defecto y con parámetros. Los constructores por defecto nos permiten inicializar las variables privadas siempre en el mismo valor que normalmente es cero, los constructores con parámetros nos permiten inicializar las variables privadas en el valor que se le pasa a los parámetros del constructor.

4.4.1.1. Constructor por defecto

```
/**************************************************************
    CONSTRUCTOR POR DEFECTO
**************************************************************/
    FuzzySet::FuzzySet()
    {
        Label = NULL;
        a = b = c = 0;
        Position = NotSet;
    }
```

4.4.1.2. Constructores con parámetros

```
/***********************************************************
    CONSTRUCTORES CON PARAMETROS
***********************************************************/
    FuzzySet::FuzzySet(std::string label)
    {
        Label = NULL;
        setLabel(label);
        a = b = c = 0;
        Position = NotSet;
    }
```

Este constructor recibe como único parámetro la etiqueta del conjunto *label* que debe ser una cadena de caracteres *string*, los demás valores *a, b, c* los inicializa en cero, y asigna a *position* el valor *Notset*.

Véase las funciones:

- *setValues(double,double,double), setLeftShouldered(), setRightShouldered(), setMiddleSet()*

```
    FuzzySet::FuzzySet(double min, double mid, double max)
    {
        Label = NULL;
        setValues(min, mid, max);
        Position = NotSet;
    }
```

Este constructor recibe como parámetros los valores *min, mid, max* tipo *double* y son inicializadas las variables *a, b, c* mediante la función *setValues,* asigna la dirección *NULL* al puntero *Label*, y la posición como *NotSet*.

Véase las funciones:

- *setLabel(string), setLeftShouldered(), setRightShouldered(), setMiddleSet().*

```
    FuzzySet::FuzzySet(std::string label,
            double min, double mid, double max)
    {
        Label = NULL;
        setValues(label, min, mid, max);
        Position = NotSet;
    }
```

Este constructor recibe como parámetros la etiqueta del conjunto, *label*, y los valores tipo *double* que definen la forma del conjunto, *min, mid, max*, para inicializar las variables privadas *a, b, c,* por medio de la función *setValues*, y asigna la posición *NotSet* al conjunto.

Véase las funciones:

- *setLeftShouldered(), setRightShouldered(), setMiddleSet()*.

```
FuzzySet::FuzzySet(std::string label,double min,
        double mid, double max, FuzzyPosition position)
{
    Label = NULL;
    setValues(label, min, mid, max);
    Position = position;
}
```

Este constructor recibe como parámetros la etiqueta del conjunto *label*, los valores tipo *double min, mid, max,* que por medio de la función *setValues* inicializa las variables privadas *a, b, c* respectivamente, también recibe la posición del conjunto *position*.

```
FuzzySet::FuzzySet(std::string label, bool isMinSet, double mid
    ,
            double max, double LowLimit)
{
    if (!isMinSet)
        {
        std::cerr << "\nERROR: Este constructor debe indicar
           que el conjunto esta a la izquierda\n";
        exit(1);
        }
    Label = NULL;
    setValues(label, LowLimit, mid, max);
    setLeftShouldered();
}
```

Este constructor recibe como parámetros la etiqueta del conjunto *label*, un valor booleano que indica que un conjunto esta ubicado a la izquierda, *isMinset*, los valores tipo *double mid* y *max*, que inicializan las variables privadas *b* y *c*, un valor tipo *double* que me indica el limite del conjunto.

Véase las funciones:

- *setRightShouldered(), setMiddleSet().*

```
FuzzySet::FuzzySet(std::string label, double min, double mid,
                   bool isMaxSet, double HighLimit)
{
    if (!isMaxSet)
        {
        std::cerr << "\nERROR: Este constructor debe indicar\n"
            ;
        exit(1);
        }
    Label = NULL;
    setValues(label, min, mid, HighLimit);
    setRightShouldered();
}
```

Recibe como parámetros la etiqueta del conjunto, los valores de *a, b* representados por *min, mid* y *HighLimit* que representa el límite del conjunto, además de un boleano indicando que el conjunto esta situado a la derecha. Asigna la posición de *RightShouldered* al conjunto.

Véase las funciones:

- *setLeftShouldered(), setMiddleSet()*.

```
/***********************************************************
    FuzzySet(FuzzySet) Constructor de copia
***********************************************************/
    FuzzySet::FuzzySet(const FuzzySet &F)
    {
        Label = NULL;
        Position = F.Position;
            if (F.Label == NULL)
                {
                    Label = NULL;
                }
                else
                {
                    std::string tempS = F.Label;
                    setValues(tempS, F.a, F.b, F.c);
                }
    }
```

Este constructor recibe como parámetro un conjunto fuzzy, para crear otro con los mismo valores de sus variables privadas.

```
FuzzySet::~FuzzySet()
{
        colorblueif (Label != NULL)
                colorbluedelete [] Label;
}
```

El destructor sirve para liberar espacio de la memoria reservada en el puntero, en este caso *label.*

4.4.2. Métodos

```
/*************************************************************
        membership(double) double
*************************************************************/
double FuzzySet::membership(double x)
{
    if (isMiddleSet() == isLeftSet() && isMiddleSet() == isRightSet
        ()){
        std::cerr << "\nERROR: La geometria del conjunto no ha sido
            definida completamente\n";
        exit(1);
        }
    double ms;
    if (x <= a || x >= c) {
        ms = 0.0;
    } else if (isLeftSet()) {
        if (x <= b)
            ms = 1.0;
        else
            ms = (c - x)/(c - b);
        } else if (isRightSet()) {
            if (x >= b)
                ms = 1.0;
            else
                ms = (x - a)/(b - a);
            } else {
                if (x == b)
                    ms = 1.0;
                else if (x < b)
                    ms = (x - a)/(b - a);
                else
                    ms = (c - x)/(c - b);
            }
            return ms;
```

```
}
```

Esta función retorna el valor de la pertenencia al conjunto de un valor *x*. La pertenencia es calculada por medio de una ecuación lineal que une los dos puntos entre los que se encuentra la variable '*x*' ya sea *AB* o *BC*, para cualquier otro caso la pertenencia es 0. Para poder usar esta función hay que definir completamente la geometría del conjunto.

véase las funciones

setValues(double,double,double), setLeftShouldered(),setRightShouldered(), setMiddleSet().

```
/*************************************************************
            setLabel(string) void
*************************************************************/
    void FuzzySet::setLabel(std::string label)
    {
        if (Label != NULL)
            delete [] Label;

        int n = label.length();
        Label = new char[n];
        const char *src = label.c_str();
        strcpy(Label, src);
}
```

Esta función asigna el valor de la variable *label* del conjunto, si existe una etiqueta anterior, es eliminada para asignar la nueva.

véase las funciones:

- *setValues(string,double,double,double)*.

```
/*************************************************************
        setValues(double,double,double) void
*************************************************************/
    void FuzzySet::setValues(double A, double B, double C)
    {
        if (A >= B || C <= B || A == C)
        {
            std::cerr << "\nERROR: La geometria es ingresada
               incongruente\n";
            exit(1);
        }
        a = A;
        b = B;
        c = C;
    }
```

Esta función asigna los valores de las variables *a, b* y *c*, imponiendo una unica restriccion de forma, es decir el valor de *a* siempre debe estar a la izquierda de *b*, y el valor de *c* siempre a la derecha de *b*.

vease la función:

- *setValues(string,double,double,double)*.

```
/*************************************************************
        setValues(string,double,double,double) void
*************************************************************/
    void FuzzySet::setValues(std::string label, double A, double B,
        double C)
    {
        setLabel(label);
        setValues(A, B, C);
    }
```

Esta función asigna el valor de las variables *label, a, b y c,* imponiendo una única restricción de forma, es decir el valor de *a* siempre debe estar a la izquierda de *b*, y el valor de *c* siempre a la derecha de *b*.

vease las funciones:

- *setLabel(string), setValues(double,double,double).*

```
/***************************************************************
            setLeftShouldered() void
***************************************************************/
    void FuzzySet::setLeftShouldered()
    {
        Position = LeftShouldered;
    }
```

Esta función asigna la posicion *LeftShoudered* al conjunto. Un conjunto con esta forma corresponde a la mitad izquierda de un trapecio.

Vease las funciones:

- *setRightShouldered(), setMiddleSet().*

```
/*****************************************************************
                setRightShouldered() void
*****************************************************************/
    void FuzzySet::setRightShouldered()
    {
        Position = RightShouldered;
    }
```

Esta función asigna la posición *RightShouldered* al conjunto. Un conjunto con esta forma corresponde a la mitad derecha de un trapecio.

Vease las funciones:

- *setLeftShouldered(), setMiddleSet().*

```
/**************************************************************
            setMiddleSet()  void
***************************************************************/
    void FuzzySet::setMiddleSet()
    {
        Position = Middle;
    }
```

Esta función asigna la posición *MiddleSet* al conjunto. Un conjunto con esta forma corresponde a un triangulo.

Vease las funciones:

- *setLeftShouldered(), setRightShouldered().*

```
/**************************************************************
                tuneA(double)  bool
***************************************************************/
bool FuzzySet::tuneA(double da)
{
    bool tune = true;
    if (a + da >= b)
    tune = false;
    else
        a = a + da;
    return tune;
}
```

Esta función sirve para hacer el proceso de afinación del conjunto imponiendo las restricciones de forma, recibe como parámetro el valor que debe ser trasladado el factor *a* del conjunto. Retorna *true* si el proceso es realizado con éxito y *false* en cualquier otro caso.

Vease las funciones:

- *tune(double,double,double), tuneB(double), tuneC(double).*

```
/**************************************************************
            tuneB(double)  bool
***************************************************************/
```

```
    bool FuzzySet::tuneB(double db)
    {
        bool tune = true;
        if (b + db <= a || b + db >= c)
            tune = false;
        else
            b = b + db;
        return tune;
    }
```

Esta función sirve para hacer el proceso de afinación del conjunto imponiendo las restricciones de forma, recibe como parámetro el valor que debe ser trasladado el factor *b* del conjunto. Retorna *true* si el proceso es realizado con exito, y *false* en cualquier otro caso.

Vease las funciones:

- *tune(double,double,double), tuneA(double), tuneC(double).*

```
/*****************************************************************
            tuneC(double) bool
*****************************************************************/
    bool FuzzySet::tuneC(double dc)
    {
        bool tune = true;
        if (c + dc <= b)
            tune = false;
        else
            c = c + dc;
        return tune;
    }
```

Esta función sirve para hacer el proceso de afinación del conjunto imponiendo las restricciones de forma, recibe como parámetro el valor que debe ser movido el factor *c* del conjunto. Retorna *true* si el proceso es realizado con éxito y *false* en cualquier otro caso.

Vease las funciones:

- *tune(double,double,double), tuneA(double), tuneB(double).*

```
/*****************************************************************
            tune(double,double,double) bool
*****************************************************************/
    bool FuzzySet::tune(double da, double db, double dc)
    {
```

```
        bool tune = true;
        if (a + da >= b + db || c + dc <= b + db)
            tune = false;
        else {
            a = a + da;
            b = b + db;
            c = c + dc;
        }
        return tune;
    }
```

Esta función sirve para hacer el proceso de afinación del conjunto imponiendo las restricciones de forma, recibe como parámetro el valor que debe ser trasladado el factor *a, b* y *c* del conjunto. Retorna *true* si el proceso es realizado con éxito y *false* en cualquier otro caso.

Vease las funciones:

- *tuneA(double), tuneB(double), tuneC(double).*

```
/*
   *******************************************************************

                 getLabel() const char*
*******************************************************************
   */
    const char *FuzzySet::getLabel()
    {
    #ifdef DEBUG
        if (Label == NULL) {
            std::cerr << "\nADVERTENCIA: La etiqueta del
                conjunto fuzzy no ha sido asignada\n";
        }
    #endif
        return Label;
    }
```

Esta función retorna una referencia constante a la variable *label*. Para usar esta función, la etiqueta debe haber sido asignada, de lo contrario retornará un puntero *NULL*.

Vease las funciones:

- *setLabel(string), setValues(string,double,double).*

```
/*****************************************************************
           getA() double
*****************************************************************/
    double FuzzySet::getA()
    {
        return a;
    }
```

Esta función retorna el valor de la variable privada *a*.
Vease las funciones:

- *setValues(double,double,double), setValues(string,double,double,double), tuneA(double).*

```
/*****************************************************************
                        getB() double
*****************************************************************/
        double FuzzySet::getB()
        {
                return b;
        }
```

Esta función retorna el valor de la variable privada *b*.
Vease las funciones:

- *setValues(double,double,double), setValues(string,double,double,double), tuneB(double).*

```
/*****************************************************************
           getC() double
*****************************************************************/
    double FuzzySet::getC()
    {
        return c;
    }
```

Esta función retorna el valor de la variable privada *c*.
Vease las funciones:

- *setValues(double,double,double), setValues(string,double,double,double), tuneC(double).*

```
/***************************************************************
                isLeftSet() bool
***************************************************************/
bool FuzzySet::isLeftSet()
{
    if (Position == LeftShouldered)
        return true;
    return false;
}
```

Esta función retorna *true* si la posición del conjunto es *LeftShouldered* y *false* en cualquier otro caso.

Vease las funciones:

- *setLeftShouldered(), setRightShouldered(), setMiddleSet(), isRightSet(), isMiddleSet().*

```
/***************************************************************
                isRightSet() bool
***************************************************************/
bool FuzzySet::isRightSet()
{
    if (Position == RightShouldered)
        return true;
    return false;
}
```

Esta función retorna *true* si la posición del conjunto es *RightShouldered* y *false* en cualquier otro caso.

Vease las funciones:

- *setLeftShouldered(), setRightShouldered(), setMiddleSet(), isLeftSet(), isMiddleSet().*

```
/***************************************************************
                isMiddleSet() bool
***************************************************************/
bool FuzzySet::isMiddleSet()
{
    if (Position == Middle)
        return true;
    return false;
}
```

Esta función retorna *true* si la posición del conjunto es *Middle* y *false* en cualquier otro caso.

Vease las funciones:

- *setLeftShouldered(), setRightShouldered(), setMiddleSet(), isRightSet(), isLeftSet().*

```
/****************************************************************
            getPosition() FuzzyPosition
****************************************************************/
    FuzzySet::FuzzyPosition FuzzySet::getPosition()
    {
        return Position;
    }
```

Esta función retorna la posición del conjunto.

Vease las funciones:

- *setLeftShouldered(), setRightShouldered(), setMiddleSet(), isRightSet(), isLeftSet(), isMiddleSet(), FuzzyPosition.*

4.4.3. Sobrecarga de operadores

4.4.3.1. Operador de flujo

```
/****************************************************************
                OPERADOR DE FLUJO
****************************************************************/
    std::ostream& operator<< (std::ostream& out, FuzzySet &F)
    {
        if (F.Label == NULL)
            out << "Etiqueta: NULL" << std::endl;
        else
            out << "Etiqueta: " << F.Label << std::endl;
        if (F.isMiddleSet())
            out << "Posicion: Medio" << std::endl;
        else if (F.isLeftSet())
            out << "Posicion: Izquierda" << std::endl;
        else if (F.isRightSet())
            out << "Posicion: Derecha" << std::endl;
        else
            out << "Posicion: NULL" << std::endl;

        out << "a = " << F.a << ", b = " << F.b << ", c = " << F.c
           << std::endl;
```

```
        return out;
    }
```

4.4.3.2. Operador de comparación

```
/*****************************************************************
            OPERADOR DE COMPARACIÓN
*****************************************************************/
bool operator== (const FuzzySet &A, const FuzzySet &B)
{
    if (A.Position != B.Position)
        return false;
    if (A.a != B.a || A.b != B.b || A.c != B.c)
        return false;

        return true;
    }
```

4.4.3.3. Operador de asignación

```
/*****************************************************************
                OPERADOR DE ASIGNACIÓN
*****************************************************************/
    FuzzySet& FuzzySet::operator =(const FuzzySet &F)
{
    a = F.a;
    b = F.b;
    c = F.c;
    Position = F.Position;
    if (F.Label == NULL)
    {
        Label = NULL;
    } else {
        std::string tempS = F.Label;
        setLabel(tempS);
    }
    return *this;
}
```

4.5. LIBRERÍA *UNIVERSE.*

Esta clase contiene los métodos y características para manejar los conjuntos que representan un universo de discurso en el modelo de NEFCLASS.

4.5.1. Constructores

4.5.1.1. Constructor por defecto

```
/*****************************************************************
              CONSTRUCTOR POR DEFECTO
*****************************************************************/
    Universe::Universe()
    {
        NumFuzzySets = 0;
        LowLimit = HighLimit = 0;
        FSets = NULL;
        restrictedOverlap = false;
        maxOverlap = 0.0;
    }
```

Este constructor inicializa las variables en cero y la variable boolena *restrictedOverlap* la iniciliza como *false*, el puntero *FSets* le pone una dirección vacia.

4.5.1.2. Constructor de copia

```
/*****************************************************************
              CONSTRUCTOR DE COPIA
*****************************************************************/
    Universe::Universe(const Universe &U)
    {
        NumFuzzySets = U.NumFuzzySets;
        LowLimit = U.LowLimit;
        HighLimit = U.HighLimit;
        restrictedOverlap = U.restrictedOverlap;
        maxOverlap = U.maxOverlap;

        if (U.FSets != NULL)
        {
            FSets = new FuzzySet[NumFuzzySets];
            for (int i = 0; i < NumFuzzySets; i++)
                FSets[i] = U.FSets[i];
        }
    }
```

Este constructor de copia, recibe como parámetro un objeto *Universe*, para inicializar las variables de un nuevo Universo con los mismos valores del Universo que se pasa como parámetro en el constructor.

4.5.1.3. Destructor

```
/*****************************************************************
                   DESTRUCTOR
*****************************************************************/
Universe::~Universe()
{
    if (FSets != NULL)
        delete [] FSets;
}
```

4.5.2. Métodos

```
/*****************************************************************
              createFuzzySets(int,double,double) void
*****************************************************************/
void Universe::createFuzzySets(int numSets, double lowValue, double
    highValue)
{
    if (FSets != NULL)
        delete [] FSets;

    if (numSets <= 0)
    {
        std::cerr << "\nERROR: El numero de conjuntos fuzzy debe
            set mayor que 0\n";
        exit(1);
    } else if (lowValue >= highValue) {
        std::cerr << "\nERROR: El mite superior del conjunto debe
            ser mayor que el inferior\n";
        exit(1);
    }

    LowLimit = lowValue;
    HighLimit = highValue;
    NumFuzzySets = numSets;
    FSets = new FuzzySet[NumFuzzySets];
    double range = HighLimit - LowLimit;
    for (int i = 0; i < NumFuzzySets; i++)
```

```
    {
        double a, b, c;
        a = LowLimit + (range*i/(NumFuzzySets + 1));
        b = LowLimit + (range*(i + 1)/(NumFuzzySets + 1));
        c = LowLimit + (range*(i + 2)/(NumFuzzySets + 1));
        FSets[i].setValues(a, b, c);

        if (i == 0)
            FSets[i].setLeftShouldered();

        else if (i == NumFuzzySets - 1)
            FSets[i].setRightShouldered();
        else
            FSets[i].setMiddleSet();
    }
}
```

Define el limite inferior y el limite superior del universo y crea tantos conjuntos *fuzzy* como sean indicados, asignandoles un traslape del 50 %. Estos conjuntos no poseen etiqueta y puede ser asignada por medio de la funcion *setFuzzyLabel.* Esta función también verifica si el puntero *FSets* apunta a una dirección que contiene un conjunto Fuzzy, en el caso de ser verdad libera memoria para asignarle una nueva dirección, igualmente verifica si el parámetro del número de conjuntos a crear es mayor que cero, si es menor que cero arroja un error.

Vease la función:

- *setFuzzyLabel(int,string).*

```
/*****************************************************************
                  setLowLimit(double) void
*****************************************************************/
    void Universe::setLowLimit(double lowValue)
    {
        if (HighLimit == LowLimit)
            LowLimit = lowValue;

        else if(lowValue < HighLimit)
            LowLimit = lowValue;
    }
```

Asigna o reasigna el valor del límite inferior del universo recibiendo el valor como parámetro, poniendo como restricción que sea menor que el límite superior.

Vease las funciones:

- *setHighLimit(double), createFuzzySets(int,double,double).*

```
/*****************************************************************
            setHighLimit(double) void
*****************************************************************/
    void Universe::setHighLimit(double highValue)
    {
        if (HighLimit == LowLimit)
            HighLimit = highValue;

        else if (highValue > LowLimit)
            HighLimit = highValue;
    }
```

Asigna o reasigna el valor del límite superior del universo, poniendo como restricción que sea mayor que el límite inferior.

Vease las funciones:

- *setLowLimit(double), setLimits(double,double), createFuzzySets(int,double,double).*

```
/*********************************************************************
            setLimits(double,double) void
*********************************************************************/
    void Universe::setLimits(double lowValue, double highValue)
    {
        setLowLimit(lowValue);
        setHighLimit(highValue);
    }
```

Recibe como parámetro el límite inferior y el límite superior del universo, que son asignados al mismo.

Vease las funciones:

- *setLowLimit(double), setHighLimit(double).*

```
/********************************************************************
            setLabels(string*) void
********************************************************************/
    void Universe::setLabels(std::string *labels)
    {
        if (FSets != NULL)
        {
            for (int i = 0; i < NumFuzzySets; i++)
                FSets[i].setLabel(labels[i]);
```

```
        }
        else {
            std::cerr << "ERROR: No se puede asignar las etiquetas,
                el numero de conjuntos no ha sido inicializado";
        }
    }
```

Recibe como parámetro un puntero que contiene las etiquetas de los conjuntos, solo sera útil si han sido inicializados los conjuntos fuzzy. Es de tener cuidado el tamaño del puntero que recibe como parámetro, ya que podria tratar de acceder a un espacio de memoria fuera del mismo.

Vease la función:

- *createFuzzySets(int,double,double).*

```
/*****************************************************************
            setRestrictedOverlap(double) void
*****************************************************************/
    void Universe::setRestrictedOverlap(double MaxOver)
    {
        if (MaxOver > 0 && MaxOver < 100)
        {
            restrictedOverlap = true;
            maxOverlap = MaxOver;
        }
    }
```

Define el valor en porcentaje del maximo traslape, y asigna *true* a la restricción del mismo.

Vease la función:

- *setUnrestrictedOverLap().*

```
/*****************************************************************
            setUnrestrictedOverLap() void
*****************************************************************/
    void Universe::setUnrestrictedOverLap()
    {
        restrictedOverlap = false;
        maxOverlap = 0.0;
    }
```

Quita la restricción del traslape máximo de los conjuntos y le asigna el valor de 0.0.

Vease la función:

- *setRestrictedOverLap().*

```
/*************************************************************
            setFuzzyLabel(int,string) void
*************************************************************/
    bool Universe::setFuzzyLabel(int numSet, std::string label)
    {
        bool fin = true;
        if (FSets == NULL)
            fin = false;

        else if(numSet >= NumFuzzySets || numSet < 0)
            fin = false;

        else
            FSets[numSet].setLabel(label);

        return fin;
    }
```

Recibe como parámetro el número del conjunto y la etiqueta del mismo. Los números de conjunto van desde 0 hasta el *NumFuzzySets* - 1. Si los cojuntos *fuzzy* no han sido creados, la función retorna *false* si el número del conjunto esta fuera del rango.
Vease la función:

- *createFuzzySets(int,double,double).*

```
/*************************************************************
            fuzzySets() FuzzySets*
*************************************************************/
    FuzzySet *Universe::fuzzySets()
    {
        if (FSets == NULL)
        {
            std::cerr << "\nPrecaucion: Los conjuntos fuzzy no han
               sido inicializados\n";
        }
        return FSets;
    }
```

Esta función retorna un puntero a la variable privada *FSets*. Es útil en la medida de que el valor retornado no es constante, por lo tanto los conjuntos pueden ser manipulados.
Vease la función:

- *createFuzzySets(int,double,double).*

```
/*************************************************************
            setInput(double) double*
*************************************************************/
    double *Universe::setInput(double Input)
    {
        if (Input > HighLimit || Input < LowLimit)
        {
            std::cerr << "\nERROR: La entrada esta por fuera del
               rango\n";
        exit(1);
        }
        double *pVect = new double[NumFuzzySets];

        if (FSets != NULL)
        {
            for (int i = 0; i < NumFuzzySets; i++)
                pVect[i] = FSets[i].membership(Input);
       }
        return pVect;
    }
```

Recibe como parámetro un valor que debe estar en el dominio del universo, y retorna un vector con las pertenencias de cada conjunto que conforman el mismo. Si esta función es usada sin crear los conjutos *fuzzy*, retornara un puntero con ceros. Esta función se debe usar con la precaución de liberar la memoria reservada despues de terminar el proceso o el uso de los datos.

Vease las funciones:

- *createFuzzySets(int,double,double), setLowLimit(double), setHighLimit(double), setLimits(double,double).*

```
/*************************************************************
        tuneSet(int,double,double,double) bool
*************************************************************/
void Universe::tuneSet(int numSet, double da, double db, double dc)
{
    if (FSets == NULL)
    {
        std::cerr << "\nERROR: Los conjuntos fuzzy no has sido
           inicializados\n";
        exit(1);
    }
```

```
    if (FSets[numSet].isLeftSet())
    {
        if (FSets[numSet].getB() + db <= FSets[numSet + 1].getB())
            FSets[numSet].tuneB(db);

    if (FSets[numSet].getC() + dc <= HighLimit)
        FSets[numSet].tuneC(dc);

    }
    else if (FSets[numSet].isRightSet())
    {
        if (FSets[numSet].getA() + da >= LowLimit)
            FSets[numSet].tuneA(da);

    if (FSets[numSet].getB() + db >= FSets[numSet - 1].getB())
        FSets[numSet].tuneB(db);
    }
    else if (FSets[numSet].isMiddleSet())
    {
        if (FSets[numSet].getA() + da >= LowLimit && FSets[numSet].
           getB() + db >= FSets[numSet - 1].getB() && FSets[numSet
           ].getB() + db <= FSets[numSet + 1].getB() && FSets[
           numSet].getC() + dc <= HighLimit)
        {
            FSets[numSet].tune(da,db,dc);

        }
        else
        {
            if (FSets[numSet].getA() + da >= LowLimit)
                FSets[numSet].tuneA(da);

        if (FSets[numSet].getB() + db >= FSets[numSet - 1].getB()
           && FSets[numSet].getB() + db <= FSets[numSet + 1].getB()
           )
            FSets[numSet].tuneB(db);

        if (FSets[numSet].getC() + dc <= HighLimit)
            FSets[numSet].tuneC(dc);
        }
    }
}
```

Esta función recibe como parametro el número del conjunto y las variaciones de cada uno de sus parametros. A pesar de que la modificación de cada uno de los con-

juntos se puede hacer por medio del puntero que retorna por la función *fuzzySets()*, este método impone otra restricción, los puntos del conjunto que al modificarse se salgan del dominio no se afinan

```
/*************************************************************
                  getMaxOverlap() double
*************************************************************/
    double Universe::getMaxOverlap()
    {
        return maxOverlap;
    }
```

Retorna el valor de la variable privada *maxOverlap*. Si la restricción de traslape no esta asignada esta función retorna 0.0

Vease las funciones:

- *setRestrictedOverlap(double), setUnrestrictedOverLap().*

```
/*************************************************************
            getNumFuzzySets() int
*************************************************************/
    int Universe::getNumFuzzySets()
    {
        return NumFuzzySets;
    }
```

Retorna el número de conjuntos fuzzy que contiene el universo.

Vease la función:

- *createFuzzySets(int,double,double).*

```
/*************************************************************
            getLowLimit() double
*************************************************************/
    double Universe::getLowLimit()
    {
        if (LowLimit == HighLimit)
        {
            std::cerr << "\nERROR: Los limites no han sido
                inicializados\n";
            exit(1);
        }
        return LowLimit;
    }
```

Retorna el valor del límite inferior del conjunto. Esta función solo se puede utilizar después de inicializar las variables de límite inferior y límite superior.
Vease las funciones:

- *createFuzzySets(int,double,double), setLowLimit(double), setHighLimit(double), setLimits(double,double).*

```
/***************************************************************
            getHighLimit() double
***************************************************************/
    double Universe::getHighLimit()
    {
        if (LowLimit == HighLimit)
        {
            std::cerr << "\nERROR:␣Los␣limites␣no␣han␣sido␣
               inicializados\n";
            exit(1);
        }
        return HighLimit;
    }
```

Retorna el valor del límite superior del conjunto. Esta función solo se puede utilizar después de inicializar las variables de límite inferior y límite superior.
Vease las funciones:

- *createFuzzySets(int,double,double), setLowLimit(double), setHighLimit(double), setLimits(double,double).*
-

4.5.3. Sobrecarga de operadores

4.5.3.1. Operador de Flujo

```
/***************************************************************
            OPERADOR DE FLUJO
***************************************************************/
    std::ostream& operator <<(std::ostream &out, const Universe &U)
    {
        out << "\n";
        if (U.FSets != NULL)
        {
            for (int i = 0; i < U.NumFuzzySets; i++)
            {
```

```
                if (U.FSets[i].getLabel() == NULL)
                    out << "Sin Etiqueta:\n";
                else
                    out << U.FSets[i].getLabel() << ":\n";

                out << U.FSets[i].getA() << "\t" << U.FSets[i].getB
                    () << "\t" << U.FSets[i].getC() << "\n";
            }
        }
        else
        {
            out << "El Universo no contiene conjuntos fuzzy" << std
                ::endl;
        }
        return out;
    }
```

4.5.3.2. Operador de asignación

```
/************************************************************
            OPERADOR DE ASIGNACIÓN
************************************************************/
    Universe& Universe::operator =(const Universe &U)
    {
        NumFuzzySets = U.NumFuzzySets;
        LowLimit = U.LowLimit;
        HighLimit = U.HighLimit;
        restrictedOverlap = U.restrictedOverlap;
        maxOverlap = U.maxOverlap;
        if (U.FSets != NULL)
        {
            FSets = new FuzzySet[NumFuzzySets];
            for (int i = 0; i < NumFuzzySets; i++)
                FSets[i] = U.FSets[i];
        }
       return *this;
    }
```

4.6. LIBRERÍA *RULENEURON*

Esta clase contiene los métodos y características para manejar reglas *fuzzy* en un modelo neuro-fuzzy NEFCLASS.

4.6.1. Constructores

4.6.1.1. Constructor por defecto

```
/*************************************************************
            CONSTRUCTOR POR DEFECTO
*************************************************************/
    RuleNeuron::RuleNeuron()
    {
        AAC = NULL;
        Antecedents = NULL;
        Activation = 1.0;
        AA = 0;
        Consecuent = 0;
        InputNeurons = 0;
        Classes = 0;
    }
```

Este constructor no recibe parámetros, inicializa los punteros con una dirección vacía, la variable *Activation* se inicializa en 1 porque el operador fuzzy que se utiliza busca el valor mínimo, por lo tanto facilita la busqueda del valor mínimo.

4.6.1.2. Constructor

```
/*************************************************************
            RuleNeuron(int) Constructor
*************************************************************/
    RuleNeuron::RuleNeuron(int NumOfInputNeurons)
    {
        setNumInputsNeurons(NumOfInputNeurons);
        AAC = NULL;
        Activation = 1.0;
        AA = 0;
        Consecuent = 0;
        Classes = 0;
    }
```

Este constructor recibe como parámetro el número de entradas y utiliza la función *setNumInputsNeurons()* para inicializar la variable *InputNeurons* las demás variables quedan igual al constructor por defecto.

4.6.1.3. Constructor

```
/*********************************************************
            RuleNeuron(int, int)
*********************************************************/
    RuleNeuron::RuleNeuron(int NumOfInputNeurons, int
       NumOfClassNeurons)
    {
        setNumInputsNeurons(NumOfInputNeurons);
        setNumClasses(NumOfClassNeurons);
        Activation = 1.0;
        AA = 0;
        Consecuent = 0;
    }
```

Este constructor recibe como parámetros el número de neuronas de entradas y el número de neuronas de salida o posibles clases del modelo, se utiliza la función *setNumInputsNeurons()* para inicializar la variable *InputNeurons*, y la función *setNumClasses* para incializar la variable *Classes.*

4.6.1.4. Constructor de copia

```
/*************************************************
                CONSTRUCTOR DE COPIA
*************************************************/
    RuleNeuron::RuleNeuron(const RuleNeuron &R)
    {
        Activation = R.Activation;
        AA = R.AA;
        Consecuent = R.Consecuent;
        InputNeurons = R.InputNeurons;
        Classes = R.Classes;
        if (R.AAC != NULL)
        {
            AAC = new double[Classes];
            for (int i = 0; i < Classes; i++)
                {
                    AAC[i] = R.AAC[i];
                }
        } else
            {
                AAC = NULL;
            }
            if (R.Antecedents != NULL)
            {
                Antecedents = new int[InputNeurons];
```

```
                for (int i = 0; i < InputNeurons; i++)
                {
                    Antecedents[i] = R.Antecedents[i];
                }
            } else
                {
                    Antecedents = NULL;
                }
    }
```

Este constructor es muy útil cuando se quiere inicializar las variables de un objeto de la clase *RuleNeuron* a partir de otro de su misma clase.

4.6.1.5. Destructor

```
/*************************************************************
                    DESTRUCTOR
*************************************************************/
    RuleNeuron::~RuleNeuron()
    {
        if (AAC != NULL)
            delete [] AAC;

        if (Antecedents != NULL)
            delete [] Antecedents;
    }
```

Este destructor se utiliza para liberar la memoria de los punteros *AAC* y *Antecedents*.

4.6.2. Métodos

```
/*************************************************************
                    setNumInputsNeurons(int) void
*************************************************************/
    void RuleNeuron::setNumInputsNeurons(int NumOfInputNeurons)
    {
        if (NumOfInputNeurons < 1)
        {
            std::cerr << "\nERROR: El numero de antecedentes es
                incorrecto\n";
            exit(1);
        }else
```

```
        {
            InputNeurons = NumOfInputNeurons;
            if (Antecedents != NULL)
                delete [] Antecedents;

            Antecedents = new int[InputNeurons];
            for (int i = 0; i < InputNeurons; i++)
                Antecedents[i] = NULL;
        }
    }
```

Recibe como parámetro el número de neuronas de entrada a la regla, el cual debe ser mayor que 1. Ajusta la variable *InputNeuron* para luego reajustar el tamaño vector de antecedentes.

Vease la función:

- *setAntecedent(int,int).*

```
/*****************************************************************
                setNumClasses(int) void
*****************************************************************/
    void RuleNeuron::setNumClasses(int NumClasses)
    {
        if (NumClasses <= 1)
        {
            std::cerr << "\nERROR: El numero de clases es
                incorrecto\n";
        } else
        {
            Classes = NumClasses;
            if (AAC != NULL)
                delete [] AAC;
            AAC = new double[Classes];
            for (int i = 0; i < Classes; i++)
            {
                AAC[i] = 0.0;
            }
        }
    }
```

Recibe como parámetro el número de clases del *model*, que debe ser mayor que 1. Luego redefine el tamaño la variable *AAC*, inicializandola en ceros.

Vease las funciones: *setClassesConsecuent(int,int), setConsecuent(int).*

```
/*****************************************************************
            setAntecedent(int,int) bool
*****************************************************************/
    bool RuleNeuron::setAntecedent(int NumOfInput, int
       NumOfSetActivated)
    {
        bool operation = true;
        if (Antecedents == NULL)
        {
            std::cerr << "\nERROR: El numero de antecedentes no ha
               sido inicializado\n";
            return false;
        }
        if (NumOfInput >= InputNeurons || NumOfInput < 0)
        {
            std::cerr << "\nERROR: El numero de antecedente esta
               fuera del rango\n";
            operation = false;
        } else
        {
            Antecedents[NumOfInput] = NumOfSetActivated;
        }
        return operation;
    }
```

Recibe como parámetro el número de la entrada, comenzando desde 0, y el número del conjunto activado para esa variable, que debe ser mayor o igual a 0. Dado el caso que la operación no se complete con éxito el valor retornado por la función es falso.

Vease las funciones:

- *getAntecedent(), setConsecuent(int), setNumInputsNeurons(int).*

```
/*****************************************************************
            setConsecuent(int) void
*****************************************************************/
    void RuleNeuron::setConsecuent(int Class)
    {
        if (Class <= 0)
        {
            std::cerr << "\nERROR: El consecuente debe ser mayor
               que cero\n";
            exit(1);
        }
```

```
        if (AAC == NULL)
        {
            std::cerr << "\nERROR: El numero de clases no ha sido
               inicializado\n";
            exit(1);
        } else if (Class > Classes)
        {
            std::cerr << "\nERROR: El consecuente es mayor que el
               numero de clases del sistema\n";
            exit(1);
        } else
        {
            Consecuent = Class;
        }
    }
```

Recibe como parámetro el número de la clase a la que corresponde la regla en el modelo, este número debe ser mayor que cero y menor o igual al número de clases del sitema. Para usar esta función es necesario inicializar el número de clases del sistema

Vease las funciones:

- *getConsecuent(), setNumClasses(int).*

```
/************************************************************
            setClassesConsecuent(int,int) void
************************************************************/
    void RuleNeuron::setClassesConsecuent(int NumClasses, int Class
       )
    {
        setNumClasses(NumClasses);
        setConsecuent(Class);
    }
```

Recibe como parámetro el número de clases del modelo, que debe ser mayor que 1 y el número del consecuente de la regla, que debe ser menor o igual al número de clases.

Vease las funciones:

- *setNumClasses(int), setConsecuent(int)*

```
/************************************************************
            setInput(double) void
************************************************************/
```

```
    void RuleNeuron::setInput(double Input)
    {
        if (Input < Activation)
            Activation = Input;
    }
```

Recibe como parámetro un valor real, para el cual se calcula la activación de la regla aplicando una *t-norma*

Vease las funciones:

- *setInput(double*), getActivation(), getActivation(double)*

```
/*****************************************************************
            setInput(double*) void
*****************************************************************/
    void RuleNeuron::setInput(double *Input)
    {
        if (Antecedents != NULL)
        {
            for (int i = 0; i < InputNeurons; i++)
            {
                setInput(Input[i]);
            }
        }
    }
```

Recibe como parámetro un puntero al primer elemento de un vector de números reales. Para usar esta función debe estar inicializado el número de neuronas de entrada a la regla, y se debe tener extremo cuidado con la longitud del vector ya que esta función podria acceder a un espacio de memoria fuera del mismo (si el vector es más corto que el número de neuronas de entrada).

Vease las funciones:

- *getActivation(), getActivation(double).*

```
/*****************************************************************
            resete() void
*****************************************************************/
    void RuleNeuron::resete()
    {
        Activation = 1.0;
        AA = 0.0;
        if (AAC != NULL)
        {
```

```
            for (int i = 0; i < Classes; i++)
            {
                AAC[i] = 0.0;
            }
        }
    }
```

Inicializa todas las variables suceptibles de cambios temporales.
Vease las funciones:

- *reseteActivation().*

```
/*************************************************************
            reseteActivation() void
*************************************************************/
    void RuleNeuron::reseteActivation()
    {
        Activation = 1.0;
    }
```

Esta función inicializa la activación de la regla con el valor 1.0.
Vease la función:

- *resete().*

4.6.3. Funciones de obtención de datos

```
/*************************************************************
            getBestClass() int
*************************************************************/
    int RuleNeuron::getBestClass()
    {
        if (AAC == NULL)
        {
            std::cerr << "\nERROR: El numero de clases no ha sido
                inicializado\n";
            exit(1);
        }
        double max = 0.0;
        int c;
        for (int i = 0; i < Classes; i++)
        {
            if (max < AAC[i])
            {
```

```
                max = AAC[i];
                c = i + 1;
            }
        }
        return c;
    }
```

Esta función retorna como valor el número de la clase con mayor activación acumulada.

Vease las funciones:

- *setConsecuent(int), setNumClasses(int).*

```
/**************************************************************
             getActivation() double
**************************************************************/
    double RuleNeuron::getActivation()
    {
        return Activation;
    }
```

Retorna el valor de la activación de la regla, se debe tener en cuenta antes de usar esta función ingresar por lo menos una entrada a la regla, ademas de no utilizar las funciones *resete()* o *reseteActivation()* antes de hacer el llamado a esta función.

Vease las funciones:

- *resete(), reseteActivation().*

```
/**************************************************************
             getActivation(double) double
**************************************************************/
    double RuleNeuron::getActivation(double Input)
    {
        setInput(Input);
        return Activation;
    }
```

Retorna el valor de activación de la regla para un valor dado, se debe tener en cuenta antes de usar esta función, ingresar por lo menos una entrada a la regla, ademas de no utilizar las funciones *resete()* o *reseteActivation()* antes de hacer el llamado a esta funcion.

Vease las funciones:

- *resete(), reseteActivation(), setInput(double), setInput(double*).*

```
/*************************************************************
            getAcumulatedActivationPerClass() double*
*************************************************************/
    double *RuleNeuron::getAcumulatedActivationPerClass()
    {
        if (AAC == NULL)
        {
            std::cerr << "\nERROR: El numero de clases no ha sido
               inicializado\n";
            exit(1);
        }
        return AAC;
    }
```

Retorna una referencia al vector de activación acumulada por clase. Antes de poder usar esta función el número de clases del sistema debe haber sido definido.

Vease las funciones:

- *getAcumulatedActivationClass(int), setNumClasses(int).*

```
/*************************************************************
            getAcumulatedActivationClass(int) double
*************************************************************/
    double RuleNeuron::getAcumulatedActivationClass(int Class)
    {
        if (AAC == NULL)
        {
            std::cerr << "\nERROR: El numero de clases no ha sido
               inicializado\n";
            exit(1);
        }
        if (Class >= Classes || Class < 0)
        {
            std::cerr << "\nERROR: El numero de clase se sale del
               rango\n";
            exit(1);
        }
        return AAC[Class];
        }
```

Recibe como parámetro el número de la clase de la cual se quiere conocer la activación acumulada, retorna el valor de dicha activación. Antes de poder usar esta función el número de clases del sistema debe haber sido definido.

Vease las funciones:

- *getAcumulatedActivationPerClass(), -setNumClasses(int).*

```
/*************************************************************
              getAntecedents() int*
*************************************************************/
    int *RuleNeuron::getAntecedents()
    {
        if (Antecedents == NULL)
        {
            std::cerr << "\nERROR: El numero de antecedentes no ha
               sido inicializado\n";
            exit(1);
        }
        return Antecedents;
    }
```

Retorna una referencia al vector de antecedentes de la regla. Antes de poder usar esta función el número de neuronas de entrada debe estar definido.

Vease la función:

- *setNumInputsNeurons(int).*

```
/*************************************************************
              getAcumulatedActivation() double
*************************************************************/
    double RuleNeuron::getAcumulatedActivation()
    {
        if (AAC == NULL)
        {
            std::cerr << "ERROR: El numero de clases del sistema no
                ha sido inicializado";
            exit(1);
        }
        AA = 0.0;
        for (int i = 0; i < Classes; i++)
        {
            if (i == Consecuent)
                AA += AAC[i];
            else
                AA -= AAC[i];
        }
        return AA;
    }
```

Retorna el valor de la activación acumulada total de la regla. Para usar esta función debe estar definido el número de clases del modelo y el consecuente, además de haber ingresado por lo menos una entrada a la regla.
Vease las funciones:

- *setConsecuent(int), setInput(double), setInput(double*), setNumClasses(int), getAcumulatedActivationPerClass().*

```
/**************************************************************
              getConsecuent()  int
**************************************************************/
    int RuleNeuron::getConsecuent()
    {
        return Consecuent;
    }
```

Retorna el número del consecuente de la regla. Para usar esta función debe estar definido el número de clases del sistema.
Vease las funciones:

- *setConsecuent(int), setNumClasses(int).*

```
/**************************************************************
              getNumInputs()  int
**************************************************************/
    int RuleNeuron::getNumInputs()
    {
        if (Antecedents == NULL)
        {
            std::cerr << "ERROR: El numero de neuronas de entrada
               no esta definido";
            exit(1);
        }
        return InputNeurons;
    }
```

Retorna el número de neuronas de entrada a la regla. Para usar esta función debe estar definido el número de neuronas de entrada.
Vease la función:

- *setNumInputsNeuron(int).*

```
/*****************************************************************
              getNumClasses() int
*****************************************************************/
    int RuleNeuron::getNumClasses()
    {
        if (AAC == NULL)
        {
            std::cerr << "ERROR: El numero de clases no esta
               definido";
            exit(1);
        }
        return Classes;
    }
```

Retorna el número de clases del sistema. Para usar esta función debe estar definido el número de clases del sistema.

Vease la función:

- *setNumClasses(int).*

4.6.4. Sobrecarga de operadores

4.6.4.1. Operador de flujo

```
/*****************************************************************
              OPAREDOR DE FLUJO
*****************************************************************/
    std::ostream& operator<< (std::ostream& out, const RuleNeuron&
       R)
    {
        if (R.Antecedents == NULL)
        {
            out << "La regla no tiene antecedente\n";
        } else
        {
            out << "La regla tiene los siguientes antecedentes\n";
            for (int i = 0; i < R.InputNeurons; i++)
                {
                    out << R.Antecedents[i] << "\t";
                }
        }
        if (R.Consecuent == 0)
        {
            out << "\nLa regla no tiene consecuente\n";
        } else
```

```
        {
            out << "\nCon consecuente:\t" << R.Consecuent << std::
                endl;
        }
        return out;
    }
```

Esta sobrecarga del operador de flujo es necesaria porque se requiere imprimir las reglas que se han formado en la arquitectura.

4.6.4.2. Operador de comparación

```
/*************************************************************
                OPAREDOR DE COMPARACION
*************************************************************/
    bool operator== (const RuleNeuron& A, const RuleNeuron& B)
    {
        if (A.InputNeurons != B.InputNeurons)
            return false;
        bool same = true;
        for (int i = 0; i < A.InputNeurons; i++)
        {
            if (A.Antecedents[i] != B.Antecedents[i])
                same = false;
        }
        return same;
    }
```

La sobrecarga del operador de comparación (==) recibe como parámetro dos objetos *RuleNeuron*, compara cada una de sus variable privadas, si son iguales retorna *true*, de lo contrario retorna *false.*

4.6.4.3. Operador de asignación

```
/*************************************************************
                OPAREDOR DE ASIGNACION
*************************************************************/
    RuleNeuron& RuleNeuron::operator =(const RuleNeuron &R)
    {
        Activation = R.Activation;
        AA = R.AA;
        Consecuent = R.Consecuent;
        InputNeurons = R.InputNeurons;
        Classes = R.Classes;
```

```
        if (R.AAC != NULL)
        {
            AAC = new double[Classes];
            for (int i = 0; i < Classes; i++)
                AAC[i] = R.AAC[i];
        }
        if (R.Antecedents != NULL)
        {
            Antecedents = new int[InputNeurons];
            for (int i = 0; i < InputNeurons; i++)
                Antecedents[i] = R.Antecedents[i];
        }
        return *this;
    }
```

La sobrecarga del operador de asignacón recibe un objeto *RuleNeuron*, y copia los valores de sus atributos al nuevo objeto, esta función es muy utilizada en el constructor de copia.

4.7. LIBRERÍA *CLASSNEURON.*

Esta librería se utiliza para crear las unidades de procesamiento de salida de la arquitectura, es útil para determinar la activación de cada una de las neuronas de salida, igualmente con esta librería podemos obtener los atributos del objeto por medio de una función especial.

4.7.1. Constructor

4.7.1.1. Constructor por defecto

```
/*************************************************************
             CONSTRUCTOR
*************************************************************/
    ClassNeuron::ClassNeuron()
    {
        Activation = 0.0;
    }
```

Este constructor se utiliza para crear un objeto de la clase *ClassNeuron* e inicializa la variable privada *Activation* en cero.

4.7.2. Métodos

```
/***********************************************************
                  setInput(double)   void
***********************************************************/
    void ClassNeuron::setInput(double Input)
    {
        if (Input > Activation)
            Activation = Input;
    }
```

Esta función nos ayuda a calcular la activación de unidad de procesamiento de salida a partir de un número de entradas a la unidad. La forma como determina la activación es escogiendo la entrada que tenga mayor valor, hay que tener encuenta que los pesos de las conexiones entre las unidades de reglas y las unidades de salida son de 1.

```
/***********************************************************
                  resete()   void
***********************************************************/
    void ClassNeuron::resete()
    {
        Activation = 0.0;
    }
```

Esta función resetea la unidad de procesamiento asignandole un valor de cero a su variable privada, esto se hace comunmente cuando se cambia de época.

```
/***********************************************************
                  getActivation()   double
***********************************************************/
    double ClassNeuron::getActivation()
    {
        return Activation;
    }
```

Esta función retorna el valor de la activación de la neurona de salida.

```
/***********************************************************
                  getActivation(double Input)   double
***********************************************************/
    double ClassNeuron::getActivation(double Input)
    {
        setInput(Input);
        return Activation;
    }
```

Esta función recibe el valor de la conexión, asigna el valor calculado a la variable *Activation* y retorna la activación.

4.8. LIBRERÍA *NETWORK*

Esta clase contiene los métodos y características que definen completamente un modelo de sistema Neuro-Fuzzy basado en el algoritmo de NEFCLASS. En esta librería se encuentran los métodos más importantes para crear una red Neuro-Fuzzy, las cuales son: *CreateRules, training, validated.*

4.8.1. Constructores

4.8.1.1. Constructor por defecto

```
/*************************************************************
               Constructor por defecto
*************************************************************/
    Network::Network()
    {
        initialize();
    }
```

El constructor por defecto llama a una función llamada *initialize()* donde se inicializan todas las variables privadas en cero y los punteros con *NULL*.

4.8.1.2. Constructor

```
/*************************************************************
                Constructor
*************************************************************/
    Network::Network(int numInputs)
    {
        initialize();
        setInputNeurons(numInputs);
    }
```

Con este constructor podemos inicializar el número de nueronas de entrada de la red por medio de la función *setInputNeurons(numInpunt)*, esta función a su vez reserva memoria para la variable privada *U*, las demás variables privadas se inicializan en cero.

4.8.1.3. Constructor

```
/**************************************************************
                Constructor
**************************************************************/
    Network::Network(int numInputs, int numClasses)
    {
        initialize();
        setInputNeurons(numInputs);
        setClassNeurons(numClasses);
    }
```

Con este constructor podemos inicializar el número de neuronas de la capa de entrada y el número de neuronas de la capa de salida, por medio del llamado de las funciones *setInputNeurons(numInputs), setClassNeurons(numClasses)* se reserva espacio para las variables privadas *U* y *CN*, donde *U* es la que contiene los Universos de cada variable de entrada y *CN* es la que contiene las classNeuron o neuronas de la capa de salida.

4.8.1.4. Constructor

```
/**************************************************************
                Constructor
**************************************************************/
    Network::Network(int numInputs,int numClasses,int maxRules,
       Pruning P)
    {
        initialize();
        setInputNeurons(numInputs);
        setClassNeurons(numClasses);
        setRuleNeurons(maxRules, P);
    }
```

Este constructor inicializa el número de neuronas de entrada, y el número de neuronas de la capa de salida o clases de la misma forma que el constructor anterior, además inicializa el número máximo de reglas *maxNumRules*, y el pruning o truncado de reglas que se le quiere aplicar a la base de Reglas *RulesPruning*.

4.8.1.5. Constructor

```
/**************************************************************
                Constructor
**************************************************************/
```

```
Network::Network(Files *handler, int maxRules, Pruning P)
{
    initialize();
    setInputNeurons(handler->getNumInputs());
    setClassNeurons(handler->getNumClasses());
    setRuleNeurons(maxRules, P);
}
```

La función de este contructor es la misma que el constructor anterior, pero los parámetros que se le deben pasar son: un manejador de archivos, el número máximo de reglas, y el pruning de las reglas.

4.8.1.6. Constructor

```
/***********************************************************
                Constructor
***********************************************************/
Network::Network(Files *handler, int maxRules, Pruning P, int
    maxEpoch)
{
    initialize();
    setInputNeurons(handler->getNumInputs());
    setClassNeurons(handler->getNumClasses());
    setRuleNeurons(maxRules, P);
    Epoch = maxEpoch;
}
```

Esta sobrecarga de constructor inicializa el número de entradas, número de neuronas de salida, máximo número de Reglas y el número de Epocas para el entrenamiento de la Red NeuroFuzzy.

4.8.2. Métodos

```
/***********************************************************
            void setInputNeurons(int numInputs)
***********************************************************/
void Network::setInputNeurons(int numInputs)
{
    INeurons = numInputs;
    if (U != NULL)
        delete [] U;
    U = new Universe[INeurons];
}
```

La función *setInputNeurons(int numInputs)*, recibe como parámetro el número de neuronas de la capa de entrada, asigna este valor a la variable *INeurons,* y reserva espacio para la variable privada que contiene los Universos *U*, esta variable es un puntero.

```
/**************************************************************
            void setClassNeurons(int classNeuron)
**************************************************************/
    void Network::setClassNeurons(int classNeuron)
    {
        CNeurons = classNeuron;
        if (CN != NULL)
            delete [] CN;
        CN = new ClassNeuron[CNeurons];
    }
```

La función *setClassNeurons(int classNeuron),* recibe como parámetro el número de neuronas de la capa de salida, asigna este valor a la variable privada *CNeurons*, y reserva espacio para la variable privada *CN* que es un puntero, esta variable contiene las neuronas de salida que son objetos que pertenecen a la clase *ClassNeuron*.

```
/**************************************************************
            void setRuleNeurons(int maxRules, Pruning P)
**************************************************************/
    void Network::setRuleNeurons(int maxRules, Pruning P)
    {
        maxNumRules = maxRules;
        RulesPruning = P;
    }
```

La función *void setRuleNeurons(int maxRules, Pruning P)*, recibe como parametros el número máximo de reglas y el tipo de truncado que se le desea hacer a las reglas, y asigna estos datos a las variables privadas *maxNumRules*, *RulesPruning*.

```
/**************************************************************
              void Network::initialize()
**************************************************************/
    void Network::initialize()
    {
        Epoch = 0;
        EpochStep = 0;
        StepByStep = false;
        INeurons = 0;
        RNeurons = 0;
```

```
        CNeurons = 0;
        NumBestRules = 0;
        RN = NULL;
        U = NULL;
        CN = NULL;
        sigmaA = 0;
        sigmaB = 0;
        sigmaC = 0;
        RulesPruning = AllRules;
        maxNumRules = 0;
    }
```

La función void *Network::initialize()* es la encargada de inicializar todas las varibles privadas en cero. Esta función es la que se utiliza en el constructor por defecto de *Network*.

```
/***************************************************************
                void createRules(Files *handler)
***************************************************************/
    void Network::createRules(Files *handler)
    {
        if (handler->getOpenMode() == Files::Training)
        {
            for (int i = 0; i < handler->getNumPatterns(); i++)
            {
                RuleNeuron tempR(INeurons,CNeurons);
                double *patt = handler->getPattern(i);
                double **M = getMP(patt);
                int *sets = getMaxMembershipSets(M);
                for (int j = 0; j < INeurons; j++)
                    tempR.setAntecedent(j, sets[j]);
                tempR.setConsecuent(handler->getClass(i));
                addRule(tempR);
                clearMatrix(M);
                delete [] patt;
            }
        } else
            std::cerr << "\nADVERTENCIA: El archivo de pratrones no
                esta abierto en modo Training\n";
    }
```

La función *void createRules(Files *handler)*, recibe como parametro la dirección de memoria del manejador de archivos, esta función es la encargada de crear las reglas, a partir del archivo de entrenamiento, cada vez que se crea una regla es guarada en una variable tipo *RuleNeuron* temporal llamada *temR*, para posteriormente

almacenarla en la variable privada *RN*, que es un puntero a objetos *RuleNeuron* por medio de la funcion *addRule(RuleNeuron)*.

```
/*************************************************************
      void setLimits(double *lowValues, double *highValues)
*************************************************************/
    void Network::setLimits(double *lowValues, double *highValues)
    {
        if (U == NULL)
        {
            std::cerr << "\nERROR: El Numero de neuronas de entrada
                no ha sido inicializado\n";
            exit(1);
        }
        for (int i = 0; i < INeurons; i ++)
        {
            U[i].setLimits(lowValues[i], highValues[i]);
        }
    }
```

La función *void setLimits(double *lowValues, double *highValues)*, recibe un puntero a *double* que contiene los límites inferiores de los universos de todas las variables de entrada, y otro puntero a *double* que contiene los límites superiores de los universos, el objetivo de esta función es definir los límites de la variable privada *U*, y es realizado por medio de la función *setLimits(lowValues[i], highValues[i])* y los punteros.

```
/*************************************************************
        setEpoches(int) void
*************************************************************/
    void Network::setNumEpoches(int maxEpoches)
    {
        Epoch = maxEpoches;
    }
```

La función *void setEpoches(int)* recibe como parametro el número de epocas en las que debe ser realizado el entrenamiento, y lo asigna a la variable *Epoches.*

- Vease las funciones : *training(Files*), training(Files*,int)*

```
/*************************************************************
        setSigma(double) void
*************************************************************/
    void Network::setSigma(double sigma)
    {
```

```
        setSigmas(sigma, sigma, sigma);
    }
```

La función *void setSigma(double)* recibe como parámetro una rata de aprendizaje la cual es asiganada a cada una de las ratas inividuales *sigmaA*, *sigmaB* y *sigmaC* por medio de la función *setSigmas(sigma, sigma, sigma)*.

- Vease la función: *setSigmas(double,double,double)*

```
/************************************************************
        setSigma(double) void
************************************************************/
    void Network::setSigmas(double sgA, double sgB, double sgC)
    {
        sigmaA = sgA;
        sigmaB = sgB;
        sigmaC = sgC;
    }
```

La función *void setSigma(double)* recibe como parámetro cada una de las ratas de aprendizaje y las asigna a *sigmaA*, *sigmaB* y *sigmaC* respectivamente.

```
/************************************************************
       addVariable(const Universe &Un) void
************************************************************/
    void Network::addVariable(const Universe &Un)
    {
        if (U == NULL)
        {
            lNeurons = 1;
            U = new Universe[lNeurons];
            U[0] = Un;
        } else {
            Universe *tempU = new Universe[lNeurons];
            for (int i = 0; i < lNeurons; i++)
                tempU[i] = U[i];
            delete [] U;
            U = new Universe[lNeurons + 1];
            for (int i = 0; i < lNeurons; i++)
                U[i] = tempU[i];
            U[lNeurons] = Un;
            lNeurons += 1;
            delete [] tempU;
        }
    }
```

La función void addVariable(const Universe &Un) recibe como parámetro un Universo, y lo agrega en la ultima posición del puntero. Esta función es muy útil en la creación de la red Neuro-Fuzzy.

```
/*****************************************************************
        addRule(RuleNeuron) void
*****************************************************************/
    void Network::addRule(const RuleNeuron &R)
    {
        if (RN == NULL)
        {
            RNeurons = 1;
            RN = new RuleNeuron[RNeurons];
            RN[0] = R;
        } else {
            if (!existRule(R))
            {
                RuleNeuron *tempR = new RuleNeuron[RNeurons];
                for (int i = 0; i < RNeurons; i++)
                    tempR[i] = RN[i];
                delete [] RN;
                RN = new RuleNeuron[RNeurons + 1];
                for (int i = 0; i < RNeurons; i++)
                    RN[i] = tempR[i];
                RN[RNeurons] = R;
                RNeurons += 1;
                delete [] tempR;
            }
        }
    }
```

La función *void Network::addRule(const RuleNeuron &R)* recibe como parámetro una regla que es agregada al final de la lista ya existente. La única restricción impuesta es que la regla no exista, ya que no tiene ningun sentido tener dos reglas identicas en el mismo modelo.

- vease: *createRules(Files*)*

```
/*****************************************************************
        existRule(RuleNeuron) bool
*****************************************************************/
    bool Network::existRule(const RuleNeuron &R)
    {
        bool ex = false;
        for (int i = 0; i < RNeurons; i++)
```

```
    {
        if (RN == NULL)
            break;
        if (RN[i] == R)
        {
            ex = true;
            break;
        }
    }
    return ex;
}
```

La función *bool existRule(const RuleNeuron &R)* recibe como parámetro una regla que es comparada con todas las reglas que en ese momento contenga el modelo. En el caso de existir alguna regla identica, la función retornara true (Verdadero), y false en el caso contrario.

- vease: *addRule(RuleNeuron)*.

```
/*************************************************************
        activateRules(double**) bool
*************************************************************/
    void Network::activateRules(double **MP)
    {
        if (RN == NULL)
        {
            std::cerr << "\nERROR: Las reglas no han sido
                inicializadas\n";
            exit(1);
        }
        for (int i = 0; i < RNeurons; i++)
        {
            bool act = true;
            for (int j = 0; j < INeurons; j++)
            {
                if (MP[j][RN[i].getAntecedents()[j]] == 0)
                {
                    act = false;
                    break;
                }
            }
            if (act)
                RN[i].triggerOn();
        }
    }
```

LA función *void Network::activateRules(double **MP)* recibe como parametro la matriz de pertenencias de los universos discurso correspondiente a cada una de las variables, con esta realiza el recorrido de todas las reglas y dispara aquellas que son determinadas por la matriz de pertenencia. Recuerde que antes de usar esta función es conveniente realizar el seteado (*RuleNeuron::resete()*) de todas las reglas, ya que no esta incluido dentro del proceso de la función.

vease: *existRule(RuleNeuron) ,pruning().*

```
/*************************************************************
        setPruning(Pruning,int) void
*************************************************************/
    void Network::setPruning(Pruning P, int maxRules)
    {
        if (P == AllRules)
        {
            RulesPruning = P;
            maxNumRules = 0;
        } else if (P == BestRules || P == BestRulesPerClass) {
            RulesPruning = P;
            maxNumRules = maxRules;
        }
    }
```

La función *void Network::setPruning(Pruning P, int maxRules)* recibe como parámetro el modelo de podado y el máximo número de reglas. Asigna el método de truncado de reglas usado por la red, el cual esta definido por el algoritmo de NEF-CLASS. El número de reglas no debe ser menor o igul a cero. Existen tres métodos, cada uno de ellos definidos por el *enum Pruning*, en el caso de usar *AllRules*, no se tendra en cuanta el valor ingresado para el máximo número de reglas el valor ingresado.

- Vease: *doPruning().*

```
/*************************************************************
        inputPatterns(Files*) void
*************************************************************/
    void Network::inputPatterns(Files *handler)
    {
        if (handler->getOpenMode() != Files::Training)
        {
            std::cerr << "El archivo no esta abierto en modo
               entrenamiento";
            exit(1);
        }
        for (int i = 0; i < handler->getNumPatterns(); i++)
```

```
        {
            double *patt = handler->getPattern(i);
            propagatePattern(patt, handler->getClass(i));
            delete [] patt;
            for (int i = 0; i < RNeurons; i++)
            {
                if (RN[i].isTrigger())
                {
                    RN[i].reseteActivation();
                    RN[i].setConsecuent(RN[i].getBestClass());
                    RN[i].adjustAcumulatedActivation(handler->
                        getClass(i), getClass());
                }
            }
        }
    }
```

La función *void Network::inputPatterns(Files *handler)* recibe como parámetro un puntero al archivo que contiene los patrones de entrenamiento, este debe de estar en modo *Training* (entrenemiento) para que exista compatibilidad de los datos. Hace pasar los patrones a través de la red NeuroFuzzy, para luego poder conocer valores como la activación acumulada por clase, activación acumulada total, etc.

Vease: *-pruning().*

```
/*********************************************************
        propagatePattern(double,int) void
*********************************************************/
    void Network::propagatePattern(double *pattern, int
        classPattern)
    {
        double **MP = getMP(pattern);
        activateRules(MP);
        for (int i = 0; i < RNeurons; i++)
        {
            if (RN[i].isTrigger())
            {
                for (int j = 0; j < INeurons; j++)
                    RN[i].setInput(MP[j][RN[i].getAntecedents()[j
                        ]]);
                RN[i].setActivationClass(classPattern);
                CN[RN[i].getConsecuent() -1].setInput(RN[i].
                    getActivation());
            }
        }
        clearMatrix(MP);
```

```
    }
```

La función void *Network::propagatePattern(double *pattern, int classPattern)* recibe como parámetro un puntero al patrón de datos y propaga este a través de la red, ademas recibe el indicativo de la clase a la que corresponde dicho patrón para realizar la cosecuente asociación.

- vease: *inputPatterns(Files)*

```
/* ************************************************************
            pruning(Files) void
************************************************************ */
    void Network::pruning(Files *handler)
    {
        if (handler->getOpenMode() != Files::Training)
        {
            std::cerr << "\nEl archivo no esta abierto en modo
               entrenamiento\n";
            exit(1);
        }
        inputPatterns(handler);
        inputPatterns(handler);
        if (RulesPruning == Network::AllRules)
        {
        /* Si el podado requiere todas las reglas, no se hace nada
           */
            for (int i = 0; i < RNeurons; i++)
                handler->saveRule(RN[i]);
        } else if (RulesPruning == Network::BestRules)
        {
            int *posRules = new int[RNeurons];
            double *act = new double[RNeurons];
            for (int i = 0; i < RNeurons; i++)
            {
                posRules[i] = i;
                act[i] = RN[i].getAcumulatedActivation();
            }
            int tempP;
            double tempA;
            for (int i = 2; i < RNeurons; i++)
            {
                for (int j = 0; j < RNeurons - 1; j++)
                {
                    if (act[j] > act[j+1])
                    {
```

```
                    tempA = act[j];
                    act[j] = act[j+1];
                    act[j+1] = tempA;
                    tempP = posRules[j];
                    posRules[j] = posRules[j+1];
                    posRules[j+1] = tempP;
                }
            }
        }
        for (int i = 0; i < maxNumRules; i++)
            handler->saveRule(RN[RNeurons-i-1]);
        RNeurons = maxNumRules;
        delete [] RN;
        delete [] posRules;
        delete [] act;
        RN = new RuleNeuron[RNeurons];
        for (int i = 0; i < RNeurons; i++)
            RN[i] = handler->readRule(i);
    }else if (RulesPruning == Network::BestRulesPerClass)
    {
        int **posBestRulesC = new int*[CNeurons];
        for (int i = 0; i < CNeurons; i++)
            posBestRulesC[i] = new int[RNeurons];
        int *pos = new int[RNeurons];
        double *activationAAR = new double[RNeurons];
        int tempP;
        double tempAAC;
        for (int i = 0; i < CNeurons; i++)
        {
            for (int j = 0; j < RNeurons; j++)
            {
                pos[j] = j;
                activationAAR[j] = RN[j].
                    getAcumulatedActivationClass(i);
            }
            for (int j = 2; j < RNeurons; j++)
            {
                for (int k = 0; k < RNeurons - 1; k++)
                {
                    if (activationAAR[k] > activationAAR[k+1])
                    {
                        tempAAC = activationAAR[k];
                        activationAAR[k] = activationAAR[k+1];
                        activationAAR[k+1] = tempAAC;
                        tempP = pos[k];
                        pos[k] = pos[k+1];
```

```
                            pos[k+1] = tempP;
                        }
                    }
                }
                for (int j = 0; j < RNeurons; j++)
                    posBestRulesC[i][j] = pos[j];
            }
            for (int i = 0; i < CNeurons; i++)
            {
                for (int j = 0; j < maxNumRules; j++)
                    handler->saveRule(RN[posBestRulesC[i][RNeurons-
                        j-1]]);
            }
            RNeurons = maxNumRules*CNeurons;
            for (int i = 0; i < CNeurons; i++)
                delete [] posBestRulesC[i];
            delete [] RN;
            delete [] posBestRulesC;
            delete [] pos;
            delete [] activationAAR;
            RN = new RuleNeuron[RNeurons];
            for (int i = 0; i < RNeurons; i++)
                RN[i] = handler->readRule(i);
        }
    }
```

La función recibe como parámetro un puntero al archivo que contenga los datos de entrenamiento, para luego hacer pasar cada uno de los patrones a través de la red, y asi hacer el truncado de reglas de acuerdo a la configuración establecida. El uso de esta función implica tener totalmente configurada la opción para el uso del archivo de reglas, para hacerlo refierase a la documentación de *Files*.

- vease: setRuleNeurons(int,Pruning)

```
/***********************************************************
            training(Files*,int) void
***********************************************************/
    void Network::training(Files *handler)
    {
        if (handler->getOpenMode() != Files::Training)
        {
            std::cerr << "\nERROR: El archivo de datos debe estar
                en modo Training\n";
            exit(1);
        }
```

```
        for (int i = 0; i < CNeurons; i++)
            CN[i].resete();
        for (int i = 0; i < RNeurons; i++)
            RN[i].resete();
        for (int i = 0; i < handler->getNumPatterns(); i++)
        {
            double *pattern = handler->getPattern(i);
            int *target = handler->getTarget(i);
            propagatePattern(pattern, handler->getClass(i));
            double *dC = calculateDeltaClass(target);
            for (int j = 0; j < RNeurons; j++)
            {
                double dR;
                double da, db, dc;
                if (RN[j].isTrigger())
                {
                    double V = RN[j].getActivation();
                    int C = RN[j].getConsecuent() - 1;
                    dR = V*(1 - V)*dC[C];
                    int Min = getMinMembershipSets(pattern, RN[j]);
                    double a, b, c;
                    int numFSet = RN[j].getAntecedents()[Min];
                    a = U[Min].fuzzySets()[numFSet].getA();
                    b = U[Min].fuzzySets()[numFSet].getB();
                    c = U[Min].fuzzySets()[numFSet].getC();
                    db = sigmaB*dR*(c-a)*sgn(RN[j].getActivation()-
                        b);
                    da = db-sigmaA*dR*(c-a);
                    dc = db+sigmaC*dR*(c-a);
                    U[Min].tuneSet(numFSet, da, db, dc);
                }
            }
            delete [] target;
            delete [] pattern;
            delete [] dC;
        }
        handler->saveWeighs(U);
    }
```

La función *void training(Files *handler)* realiza el proceso de entrenemiento, recibe como parámetro el puntero al *handler* de archivo y el máximo número de epocas de entrenamiento, es de tener en cuenta que el archivo con los datos de entrenamiento debe estar abierto en modo *training*.

- vease: *setNumEpoches(int), training(Files*).*

```
/*****************************************************************
            training(Files*,int) void
*****************************************************************/
    void Network::training(Files *handler, int maxEpoches)
    {
        setNumEpoches(maxEpoches);
        training(handler);
    }
```

La función *void training(Files *handler, int maxEpoches)* realiza el proceso de entrenamiento, recibe como parámetro el puntero al *handler* de archivo y el máximo número de epocas de entrenamiento, es de tener en cuenta que el archivo con los datos de entrenamiento debe estar abierto en modo *training*.

- Vease: *setNumEpoches(int), training(Files*)*

```
/*****************************************************************
            validate(double*,int)
*****************************************************************/
    bool Network::validate(double *pattern, int classPattern)
    {
        if (RN == NULL)
        {
            std::cerr << "\nERROR: Las reglas no han sido
               inicializadas\n";
            exit(1);
        }
        if (CN == NULL)
        {
            std::cerr << "\nERROR: Las clases no han sido
               inicializadas\n";
            exit(1);
        }
        bool val = true;
        for (int i = 0; i < RNeurons; i++)
            RN[i].resete();
        for (int i = 0; i < CNeurons; i++)
            CN[i].resete();
        propagatePattern(pattern, classPattern);
        aproximate();
        int C = getClass();
        if (C != classPattern)
        val = false;
        return val;
    }
```

La función *bool validate(double *pattern, int classPattern)* recibe como parámetro un patrón de datos y la clase asociada al mismo, para retornar *true* siendo positiva la clasificación del mismo.
Vease: *training(Files*), training(Files*,int)*.

```
/****************************************************************
                   getClass() int
****************************************************************/
     int Network::getClass()
     {
          double max = CN[0].getActivation();
          int pos = 1;
          for (int i = 1; i < CNeurons; i++)
          {
               if (max < CN[i].getActivation())
               {
                    max = CN[i].getActivation();
                    pos = i + 1;
               }
          }
          return pos;
     }
```

La función *int Network::getClass()* retorna la posición de la *ClassNeuron* con la mayor activación.

- Vease: *propagatePattern(double*,int), inputPatterns(Files)*.

```
/****************************************************************
                   aproximate() void
****************************************************************/
     void Network::aproximate()
     {
          double max = CN[0].getActivation();
          int pos = 0;
          for (int i = 1; i < CNeurons; i++)
          {
               if (max < CN[i].getActivation())
               {
                    max = CN[i].getActivation();
                    pos = i;
               }
          }
          for (int i = 0; i < CNeurons; i++)
          {
```

```
            if (i == pos)
                CN[i].setInput(1);
            else
                CN[i].resete();
        }
    }
```

La función void Network::aproximate() realiza la tarea de redondear a 1 la ClassNeuron con mayor activación.

- Vease: *validate(double*,int).*

```
/****************************************************************
                calculateDeltaClass(int*) double*
****************************************************************/
    double* Network::calculateDeltaClass(int* target)
    {
        double *dC = new double[CNeurons];
        for (int i = 0; i < CNeurons; i++)
            dC[i] = (double)target[i] - CN[i].getActivation();
        return dC;
    }
```

Esta función recibe como parámetro el vector objetivo y calcula la distacia entre este y la salida de la red, retornando un vector que contiene el diferencial o error. Hay que liberar la memoria almacenada por esta función luego de usar los datos.

- Vease: *inputPatterns(Files*).*

```
/****************************************************************
            getMaxMembershipSets(double**) int*
****************************************************************/
    int* Network::getMaxMembershipSets(double **MP)
    {
        int *sets = new int[INeurons];
        for (int i = 0; i < INeurons; i++)
        {
            double temp = 0;
            for (int j = 0; j < U[i].getNumFuzzySets(); j++)
            {
                if (MP[i][j] > temp)
                {
                    temp = MP[i][j];
                    sets[i] = j;
```

```
                }
            }
        }
        return sets;
    }
```

La función *int* getMaxMembershipSets(double **MP)* recibe como parámetro la matriz de pertenencias y retorna el número de los conjuntos fuzzy con mayor pertenencia en cada universo

- Vease: *getMP(double*), createRules(Files*)*

```
/*************************************************************
            getMInMembershipSets(double**,RuleNeuron) int
*************************************************************/
    int Network::getMinMembershipSets(double **MP, RuleNeuron R)
    {
        double Min = MP[0][R.getAntecedents()[0]];
        int Pos = 0;
        for (int i = 1; i < INeurons; i++)
        {
            if (MP[i][R.getAntecedents()[i]] < Min)
            {
                Min = MP[i][R.getAntecedents()[i]];
                Pos = i;
            }
        }
        return Pos;
    }
```

La función *int getMInMembershipSets(double**,RuleNeuron)* recibe como parámetro la matriz de pertenencias y una regla, con los cuales calcula cual universo es el que tiene menor grado de activación.

- Vease: *getMaxMembershipSets(double*), getMinMembershipSets(double*,RuleNeuron).*

```
/*************************************************************
            getMInMembershipSets(double*,RuleNeuron) int
*************************************************************/
    int Network::getMinMembershipSets(double *Pattern, RuleNeuron R
        )
    {
        double **MP = getMP(Pattern);
        int Min = getMinMembershipSets(MP, R);
```

```
        clearMatrix(MP);
        return Min;
    }
```

La función *int getMinMembershipSets(double *Pattern, RuleNeuron R)* recibe como parámetro un patrón de datos y una regla, con los cuales calcula cual universo es el que tiene menor grado de activación.

- Vease: *getMaxMembershipSets(double*), getMinMembershipSets(double**,RuleNeuron).*

```
/*************************************************************
            getMP(double*) double**
*************************************************************/
    double** Network::getMP(double *inpt)
    {
        double **MP = new double*[INeurons];
        for (int i = 0; i < INeurons; i++)
            MP[i] = U[i].setInput(inpt[i]);
        return MP;
    }
```

La función *double** Network::getMP(double *inpt)* recibe como parámetro un puntero que contiene un patrón de datos, el cual es usado para calcular las pertenencias de cada uno de los conjuntos en cada uno de los universos , esta función primero almacena memoria para cada una de las entradas, luego (usando la función s*etInput*) implicitamente reserva memoria para cada uno de los vectores que contienen las pertenencias de los universos. Se recomienda liberar la memoria reservada por esta función por medio del metodo *clearMatrix()*. Esto tambien puede hacerse manualmente pero teniendo la precaución de crear el método correcto.

- vease: *clearMatrix(double**)*

```
/*************************************************************
            reseteNeurons() void
*************************************************************/
    void Network::reseteNeurons()
    {
        if (RN == NULL)
            std::cerr << "\nADVERTENCIA: La neuronas de reglas no
               han sido inicializadas\n";
        else {
            for (int i = 0; i < RNeurons; i++)
                RN[i].resete();
        }
```

```
        if (CN == NULL)
            std::cerr << "\nADVERTENCIA: La neuronas de clases no
               han sido inicializadas\n";
        else {
            for (int i = 0; i < CNeurons; i++)
                CN[i].resete();
        }
    }
```

La función *void Network::reseteNeurons()* resetea tanto las neuronas de Reglas como las neuronas de Clases.

- Vease: *reseteUniverses(), resete().*

```
/*****************************************************************
            reseteUniverses() void
*****************************************************************/
    void Network::reseteUniverses()
    {
        if (U == NULL)
            std::cerr << "\nADVERTENCIA: Los universos no han sido
               inicializados\n";
        else {
            for (int i = 0; i < INeurons; i++)
                U[i].reseteFuzzySets();
        }
    }
```

La función void *reseteUniverses()* resetea los universos, correspondientes a los pesos sinápticos.

- Vease: *reseteNeurons(), resete().*

```
/*****************************************************************
            resete() void
*****************************************************************/
    void Network::resete()
    {
        reseteNeurons();
        reseteUniverses();
    }
```

La función *void Network::resete()* resetea los universos y las neuronas de la red.

- Vease: *reseteNeurons(), reseteUniverses()*.

```
/*****************************************************************
            clearMatrix(double**) void
*****************************************************************/
    void Network::clearMatrix(double **M)
    {
        for (int i = 0; i < lNeurons; i++)
        {
            delete [] M[i];
        }
        delete [] M;
    }
```

La función *void Network::clearMatrix(double **M)* libera la memoria reservada para una matriz, esta función esta especialmente diseñada para liberar la memoria del metodo *getMP*.

Vease: *getMP(double*).*

```
/*****************************************************************
            clearMatrix(double**) void
*****************************************************************/
    int Network::sgn(double Value)
    {
        if (Value > 0)
            return 1;
        else if (Value < 0)
            return -1;
        else
            return 0;
    }
```

La función *int sgn(double)* recibe como parámetro un número real y de acuerdo a su valor (mayor o menor que cero) calcula la salida del método.

Vease: *training(Files*)*

4.9. CÓMO PROGRMAR CON LAS LIBRERÍAS?

En esta sección veremos como crear cada uno de los objetos que componen una red Neuro-Fuzzy, esto con el fin de demostrar el potencial que tiene las librerías. Primero veremos un ejemplo de como crear un conjunto fuzzy.

```
1 /* Ejemplo para crear un Conjuntos fuzzy*/
2 string s = "Conjunto Fuzzy";
3 double a = 1.1;
4 double b = 2.2;
5 double c = 3.3;
```

```
 6 FuzzySet F(a,b,c);
 7 F.setLabel(s);
 8 F.setMiddleSet();
 9 cout << F.membership(2.6) << endl;
10 F.tuneC(-0.5);
11 cout << F.membership(2.6) << endl;
12 cout << F;
```

En la linea 2 se crea un string con el nombre de la etiqueta que se le desea colocar al conjunto. En las lineas 3,4,5. se tres objetos tipo double que van a dar la forma del conjunto, asumiendo que el conjunto tiene una forma triangular. En la linea 6 se llama un constructor donde se crea un conjunto fuzzy *F* pasando los parámetros a, b, c de un conjunto triangular[4]. En la linea 7 de llama la función *setLabel* que le asigna la etiqueta al conjunto *F*. En la linea 8 se llama la función le asigna al conjunto *F* el tipo de conjunto, en este caso es un conjunto que se ubica en la mitad de un Universo. En la linea 9 se esta imprimiendo la pertenencia de 2,6 al conjunto *F*. En la linea 10 se esta modificando el valor *c* del conjunto disminuyéndolo en 0.5. En la linea 11 se imprime nuevamente la pertenencia de 2.6 al conjunto para verificar el cambio que forma que se hizo al conjunto. Y finalmente en la 12 se imprime las principales características del conjunto F.

Ahora veremos un ejemplo en el que crearemos un Universo compuesto de 3 conjuntos fuzzy.

```
 1 /* Ejemplo para crear un Universo de conjuntos fuzzy */
 2 int numFuzzySet = 3;
 3 string Set1 = "small";
 4 string Set2 = "meduium";
 5 string Set3 = "large";
 6 Universe U;
 7 U[i].createFuzzySets(numFuzzySet,0,4);
 8 U[i].setFuzzyLabel(0,Set1);
 9 U[i].setFuzzyLabel(1,Set2);
10 U[i].setFuzzyLabel(2,Set3);
```

En las 4 primares lineas se definen el número de conjuntos fuzzy que va a tener el Universo y sus respectivas etiquetas. En la linea 5 se crea el universo U llamando un constructor por defecto. En la linea 6 utilizamos una función que crea los conjuntos fuzzy para el Universo U teniendo el numero de conjuntos fuzzy, el límite inferior del Universo que en esta caso es 0 y el limeta superior 4, esta función crea los conjuntos de tal manera que queden bien distribuidos en el Universo. En las lineas 7, 8, 9 se definen las etiquetas de cada uno de los conjuntos del Universo U.

A continuación veremos como crear una Regla.

```
1 /* Ejemplo para crear un regla fuzzy */
2 double *vec = new double[2];
3 vec[0] = 0.5;
```

[4] Ver la sección 3.1 pagina 21

```
4  vec[1] = 0.2;
5  RuleNeuron R;
6  R.setNumInputsNeurons(2);
7  R.setAntecedent(0,1);
8  R.setAntecedent(1,0);
9  R.setNumClasses(4);
10 R.setConsecuent(3);
11 R.setInput(vec);
12 cout << R << R.getActivation();
```

En las 4 primeras lineas se creo un vector que va a simular las entradas a una red Neuro-Fuzzy. En la linea 5 se crea una reglas llamando el constructor por defecto. En la linea numero 6 se llama una función que define el numero de entradas a la red, esto con el fin de saber el antecedente cuantas condiciones debe tener. En las lineas 7 y 8 se define el antecedente llamando la función setAntecedent que recibe como primer parámetro el número de la entrada y como segundo parámetro en número del conjunto fuzzy, suponiendo que están ordenados de izquierda a derecha en el Universo, es decir el conjunto que esta a la izquierda del universo es el conjunto número 0, En la linea numero 9 se define el consecuente que es el número de la clase. En al linea número 10 se ingresan el valor de cada conexión para poder obtener una activación de la Reglas, y finalmente en la linea número 11 se imprime la activación de la Regla.

A continuación veremos como crear una Neurona de salida.

```
1  /* Ejemplo para crear una neurona de salida */
2  ClassNeuron N;
3  N.setInput(0,5);
4  N.setInput(0,7);
5  cout << "la activación de la neurona es: " << N.getActivation() << "\n";
```

En la linea número 2 se crea la neurona de salida con un constructor por defecto, en las lineas 3 y 4 se calcula la activación de la neurona ingresando datos numéricos que los proporciona las conexiones, el objetivo de esta función es llamarla tantas veces como el numero de conexiones que tenga la neurona con el fin de obtener el valor mayor como activación, y en la linea número 5 se imprime el valor de la activación de la neurona.

5. CÓMO UTILIZAR NEFCLASS-Q?

NEFCLASS-Q es la herramienta que se desarrolló en el presente proyecto, el cual es un software destinado a la creación, entrenamiento y validación de de una red Neuro-Fuzzy para la clasificación de patrones de datos.

NEFCLASS-Q tiene dos formas de funcionar, la primera es por medio de una interfaz, la cual facilitara el proceso de creación de las red Neuro-Fuzzy que será explicada en este cépitulo, y la segunda es ustizar las librerías de NEFCLASS-Q para crear un agente inteligente por medio de un pequeño codigo en C++ y será explicado en el siguiente cápitulo.

NEFCLASS-Q fue desarrollado teniendo como base el modelo de un perceptrón fuzzy de 3 capas que fue fruto de los estudios hechos por los Doctores Rudolf Kruse y Detleft Nauck de la Universidad de Magdeburg Alemania en el año 1994.

NEFCLASS-Q crea una Red Neuro-Fuzzy que es entrenado a partir de un conjunto de datos almacenados en un archivo de texto, donde cada patrón de datos pertenece a una clase determinada. El primer paso importante que hace la herramienta es la creación de Reglas Fuzzy por medio de la propagación de los datos de entrenamiento, posteriormente estas reglas son optimizadas por medio del aprendizaje de los parámetros de los conjuntos Fuzzy que conforman estas reglas. Después de que el sistema es entrenado se prosigue a la etapa de validación del entrenamiento, donde se puede evaluar el entrenamiento de la red, en caso de que los resultados se quieran mejorar se podrá cambiar algunos parámetros de entrenamiento para obtener mejores resultados.

Después de esto pasamos a la etapa de clasificación donde los patrones del archivo de clasificación son cargados y clasificados.

Para obtener más información acerca del Perceptrón Fuzzy dirigirse al cápitulo 3 de la tesis. A continuación se va a explicar cómo crear un sistema Neuro-Fuzzy con la herramienta NEFCLASS-Q.

Antes de empezar a crear la red, primero se debe crear los archivos de entrenamiento, validación y clasificación. Para ello se tienen formatos específicos.

Para configurar el archivo de entrenamiento como el de la figura 5.1, en la primera línea se debe colocar el número de patrones, el número de variables y el numero clases separadas por un espacio, en el caso de la figura 5.1 el número de patrones es 75, el número de variables es 4 y el número de clases es 3. De las líneas 2 a la 5 se escriben los dominios de cada variable, es decir los valores mínimo y máximo que puede tomar cada variable, igualmente separados por un espacio, y de la línea 6 en adelante están los patrones que contiene 5 columnas, las 4 primeros son las entradas a la red y la columna cinco es la clase a la que pertenece el patrón, por ejemplo el patrón de la línea 6 pertenece a clase 1.

El archivo de validación es el que se encarga de proporcionar los datos para evaluar el entrenamiento de la red, la estructura de este archivo es el de la figura

Figura 5.1: Archivo de entrenamiento

```
 1 75 4 3
 2 4 8
 3 2 5
 4 1 7
 5 0 3
 6 5.1  3.5  1.4  0.2  1 0 0
 7 7.0  3.2  4.7  1.4  0 1 0
 8 6.3  3.3  6.0  2.5  0 0 1
 9 4.9  3.0  1.4  0.2  1 0 0
10 6.4  3.2  4.5  1.5  0 1 0
11 5.8  2.7  5.1  1.9  0 0 1
12 4.7  3.2  1.3  0.2  1 0 0
13 6.9  3.1  4.9  1.5  0 1 0
14 7.1  3.0  5.9  2.1  0 0 1
15 4.6  3.1  1.5  0.2  1 0 0
16 5.5  2.3  4.0  1.3  0 1 0
17 6.3  2.9  5.6  1.8  0 0 1
18 5.0  3.6  1.4  0.2  1 0 0
19 6.5  2.8  4.6  1.5  0 1 0
```

Fuente: Autores

5.2. La primera línea contiene el número de variables de entrada, la segunda línea contiene el número de patrones de datos existente en el archivo, de la línea 3 en adelante contiene los patrones de datos con las respectivas clases.

Figura 5.2: Archivo de validación

```
 1 4
 2 75
 3 5.0  3.0  1.6  0.2  1 0 0
 4 6.6  3.0  4.4  1.4  0 1 0
 5 7.2  3.2  6.0  1.8  0 0 1
 6 5.0  3.4  1.6  0.4  1 0 0
 7 6.8  2.8  4.8  1.4  0 1 0
 8 6.2  2.8  4.8  1.8  0 0 1
 9 5.2  3.5  1.5  0.2  1 0 0
10 6.7  3.0  5.0  1.7  0 1 0
11 6.1  3.0  4.9  1.8  0 0 1
12 5.2  3.4  1.4  0.2  1 0 0
13 6.0  2.9  4.5  1.5  0 1 0
14 6.4  2.8  5.6  2.1  0 0 1
15 4.7  3.2  1.6  0.2  1 0 0
16 5.7  2.6  3.5  1.0  0 1 0
17 7.2  3.0  5.8  1.6  0 0 1
```

Fuente: Autores

El archivo de clasificación es similar al de validación con la única diferencia que no se escriben a que clase pertenece cada patrón de datos, como se puede ver en la figura 5.3

Figura 5.3: Archivo de clasificación

```
1  4
2  75
3  5.0  3.0  1.6  0.2
4  6.6  3.0  4.4  1.4
5  7.2  3.2  6.0  1.8
6  5.0  3.4  1.6  0.4
7  6.8  2.8  4.8  1.4
8  6.2  2.8  4.8  1.8
9  5.2  3.5  1.5  0.2
10 6.7  3.0  5.0  1.7
11 6.1  3.0  4.9  1.8
12 5.2  3.4  1.4  0.2
13 6.0  2.9  4.5  1.5
14 6.4  2.8  5.6  2.1
15 4.7  3.2  1.6  0.2
```

Fuente: Autores

Después de tener bien configurados los archivos de entrenamiento, validación y clasificación, podemos abrir el programa NEFCLASS-Q y nos aparecerá la ventana de la figura 5.4 .

Figura 5.4: Ventana inicial NEFCLASS-q

Untitle.nfc - Neuro-Fuzzy Clasificator
Archivo Visualizar Ayuda
CONFIGURACION DE LA RED NEURO-FUZZY
INFORMACION GENERAL DE LA RED
Nombre de la Red:
Directorio de trabajo:
Numero de Entradas:
Numero Maximo de Reglas:
Numero de Clases del Sistema:
Numero de conjuntos iguales por Universo:
NOMBRE DE LAS CLASES
Editar
CONFIGURACION DEL APRENDIZAJE
All Rules Best Rules BestRules per class
Numero de mejores Reglas 20
Configurado Configurar
ETIQUETAS FUZZY PARA LOS UNIVERSOS
Agregar
Editar
Subir
Bajar
Borrar
CONFIGURACION DEL ENTRENAMIENTO
Numero de epocas 500
Ratas de Aprendizaje Usar Única
a 0,00100 c 0,00100
b 0,00100 u 0,00100
Configurado Configurar

Fuente: Autores

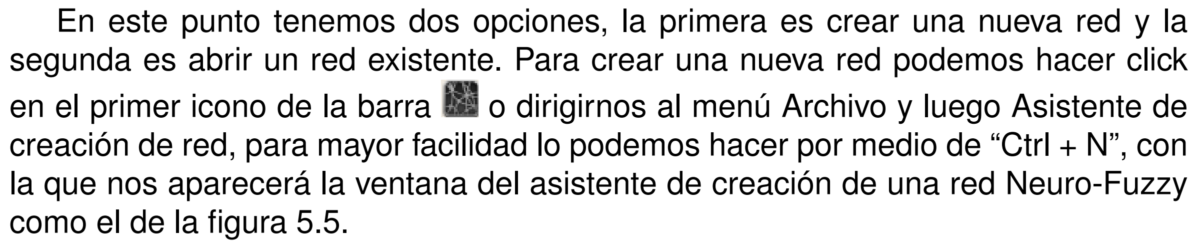
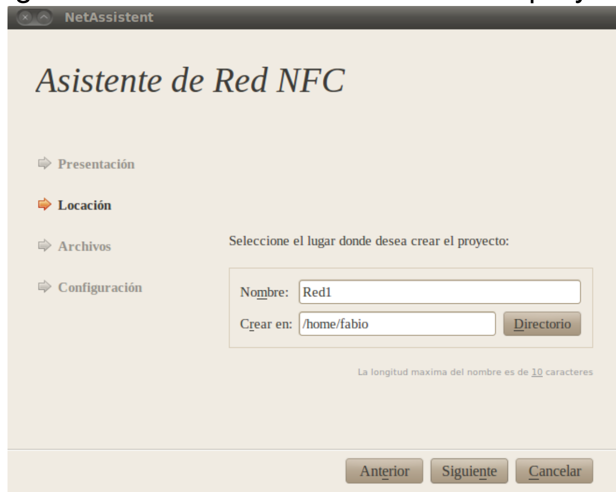
En este punto tenemos dos opciones, la primera es crear una nueva red y la segunda es abrir un red existente. Para crear una nueva red podemos hacer click en el primer icono de la barra  o dirigirnos al menú Archivo y luego Asistente de creación de red, para mayor facilidad lo podemos hacer por medio de “Ctrl + N”, con la que nos aparecerá la ventana del asistente de creación de una red Neuro-Fuzzy como el de la figura 5.5.

Figura 5.5: Primera página del asistente de creación de red

Fuente: Autores

En esta ventana recibimos información acerca de lo que se puede realizar con el asistente y damos click en el botón siguiente. Nos aparece una ventana como la de la figura 5.6 en la que le se debe asignar un nombre al proyecto que vamos a crear e igualmente podemos seleccionar el directorio en el que se va a guardar el proyecto.

Figura 5.6: Creación del directorio del proyecto



Fuente: Autores

Después de haber seleccionado la ubicación del proyecto le damos siguiente y aparece una ventana como la de la figura 5.7 en la se cargan los archivos de entrenamiento, validación y clasificación, para ello le debemos dar click en el botón archivos y seleccionamos los tres archivos, en el caso que no se cargue el archivo de clasificación o validación, no se podrá realizar dichos procedimientos, pero si es importante siempre cargar el archivo de entrenamiento.

Figura 5.7: Carga de archivos

Fuente: Autores

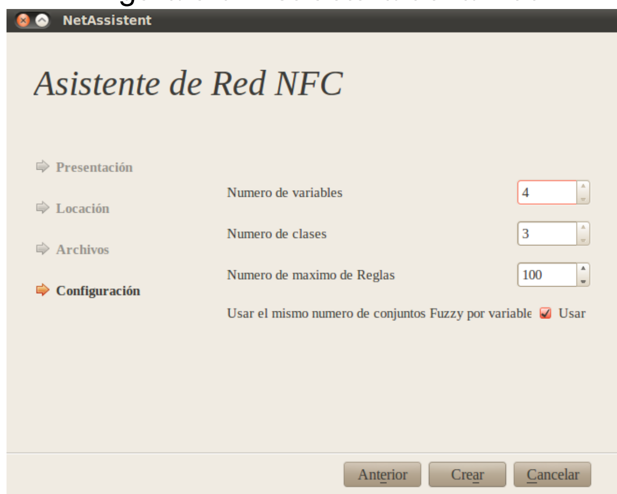
Cuando los nombres de los archivos son cargados en la ventana pasmos a dar doble click a cada uno y seleccionar el tipo de archivo, ya sea entrenamiento, validación o clasificación como muestra la figura 5.8.

Figura 5.8: Definición tipos de archivos cargados

Fuente: Autores

A continuación se puede dar click en siguiente y nos aparece la ventana de la figura 5.9, en esta se define automáticamente el número de variables y el número de clases ya que son obtenidas del archivo de entrenamiento, el usuario debe definir el número máximo de reglas que se puede crear y definir si desea que cada variable tenga el mismo número de conjuntos fuzzy por universo o variable, luego damos click en el botón crear y se guardan los datos del proyecto.

Figura 5.9: Estructura de la Red



Fuente: Autores

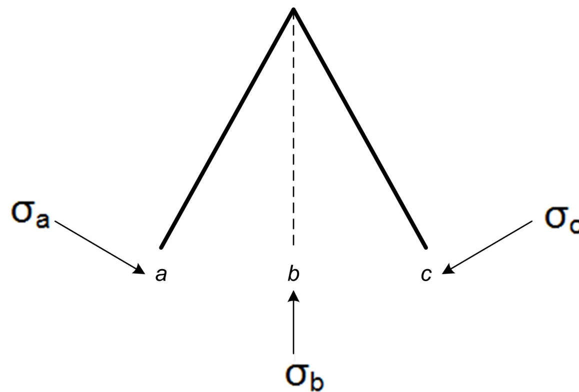
Cuando damos click en crear nos va a salir un mensaje de que el proyecto ha sido creado exitosamente, le damos click en el botón OK del mensaje y pasamos a la ventana de configuración de parámetros de entrenamiento y aprendizaje de la red como muestra la figura 5.10.

Figura 5.10: Configuración de la Red Neuro-Fuzzy

Fuente: autores

Esta ventana está dividida en varias secciones, la primera es la información general de la red donde se encuentra todos los datos definidos en el asistente de la red y no son modificables. En la configuración del aprendizaje podemos seleccionar un método de pruning de reglas y el número de mejores reglas en caso de seleccionar el pruning de mejores reglas o mejores reglas por clase debemos dar click en el botón configurar para aplicarlos. En la configuración del entrenamiento se define el número de épocas y las ratas de aprendizaje de los pesos, aquí se puede aplicar el mismo valor de la rata para los 3 puntos del conjunto o definirlos de manera independiente, en la figura se muestra en qué puntos del conjunto van a actuar,

Figura 5.11: Puntos donde actúan los coeficientes de aprendizaje



Fuente: Autores

igualmente se debe dar click en el botón configurar para aplicarlos a la red. En

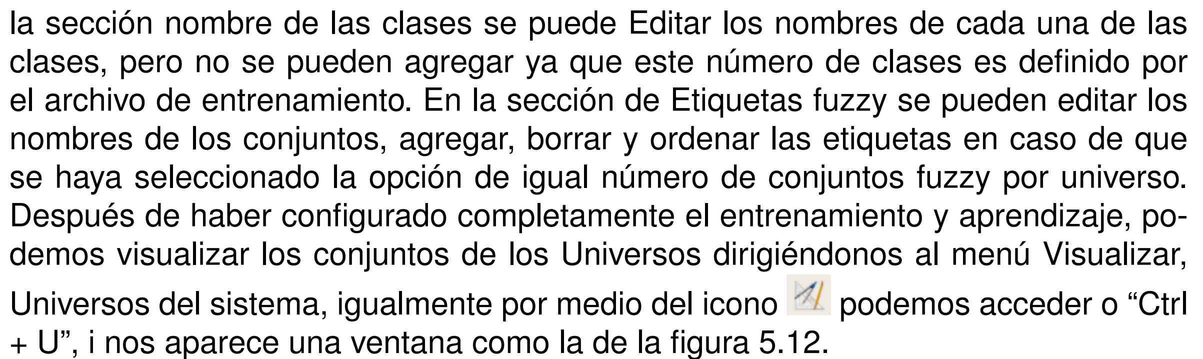
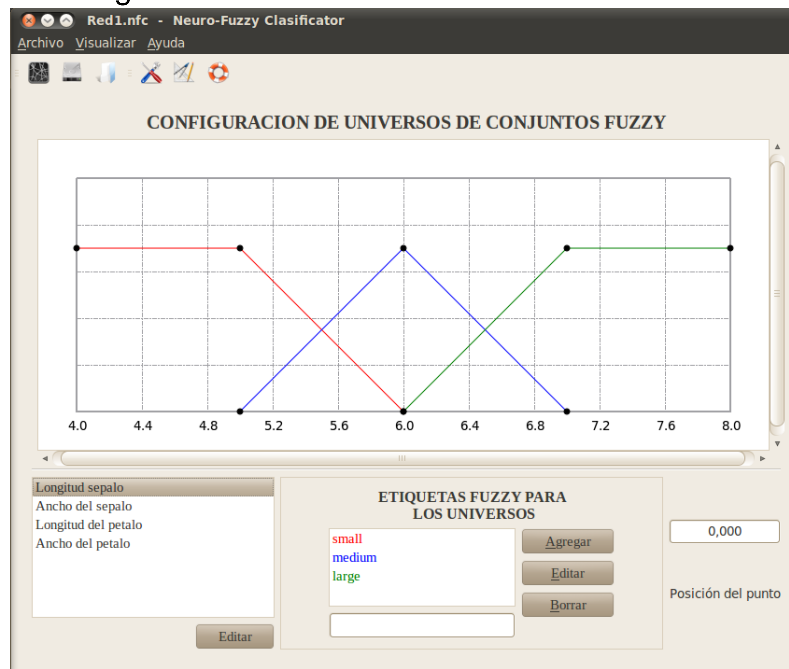
la sección nombre de las clases se puede Editar los nombres de cada una de las clases, pero no se pueden agregar ya que este número de clases es definido por el archivo de entrenamiento. En la sección de Etiquetas fuzzy se pueden editar los nombres de los conjuntos, agregar, borrar y ordenar las etiquetas en caso de que se haya seleccionado la opción de igual número de conjuntos fuzzy por universo. Después de haber configurado completamente el entrenamiento y aprendizaje, podemos visualizar los conjuntos de los Universos dirigiéndonos al menú Visualizar, Universos del sistema, igualmente por medio del icono  podemos acceder o “Ctrl + U”, i nos aparece una ventana como la de la figura 5.12.

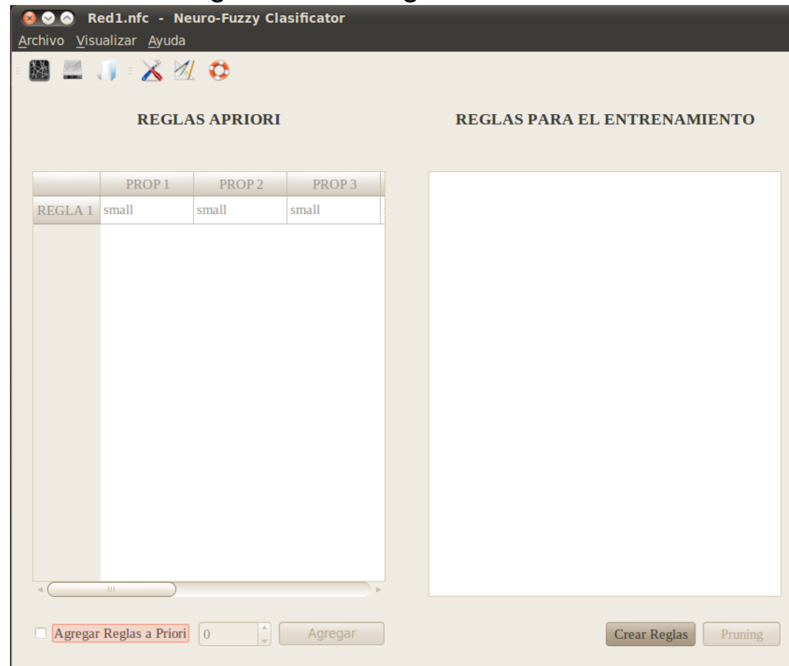
Figura 5.12: Visualizar Universos del sistema



Fuente: Autores

En esta ventana se puede ajustar manualmente los conjuntos arrastrando los puntos en la gráfica, se puede editar el nombre de las variables, igualmente si se quiere que un conjunto tenga más conjuntos fuzzy se puede agregar, borrar y editar las etiquetas de los mismos. Después de haber definido la forma de los conjuntos, se puede realizar el entrenamiento, para ello necesitamos ir a Visualizar, Reglas del sistema o con “Ctrl + R” y nos aparece una ventana como la de la figura 5.13.

Figura 5.13: Reglas del sistema



Fuente: Autores

En esta ventana se puede agregar conocimiento previo y crear reglas, para agregar el conocimiento previo debe seleccionar la casilla Agregar Reglas a priori que esta en la parte inferior izquierda como muestra la figura 5.14 y colocamos las reglas que deseamos agregar.

Figura 5.14: Agregar Reglas Fuzzy

Fuente: Autores

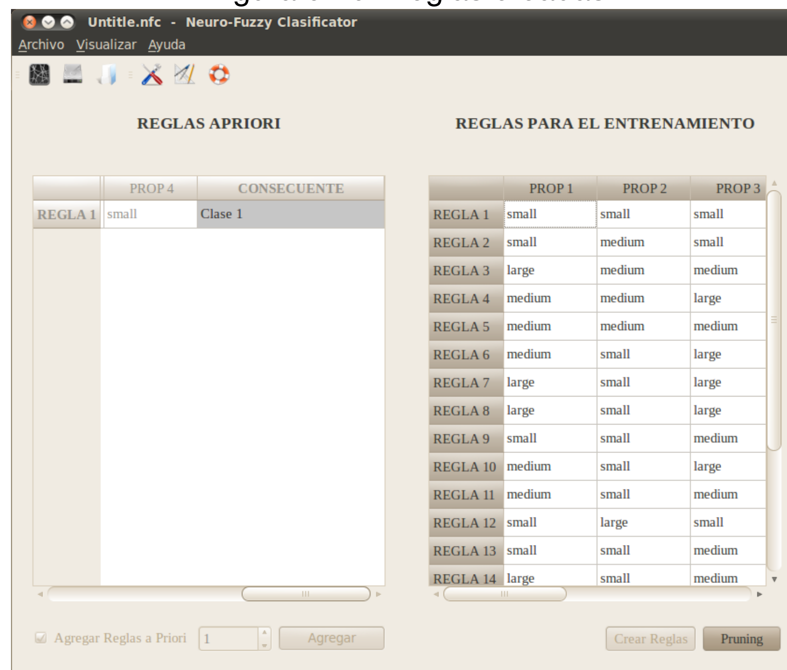
Con esto se nos habilita la tabla con la que podemos editar las reglas, para ello debemos dar doble click en la proposición que deseamos editar y seleccionamos el valor lingüístico, en caso de que no se quiera tener en cuenta esa proposición se selecciona el ítem “Ninguno”, como muestra la figura 5.15. Cuando se tenga la reglas se debe dar click en el botón Agregar.

Figura 5.15: Selección proposición

Fuente: Autores

Después de haber agregado las reglas a priori se debe crear las reglas, para ello debemos dar click en el botón “Crear reglas” y posteriormente al botón “Pruning”, las reglas definitivas son mostradas como en la figura 5.16.

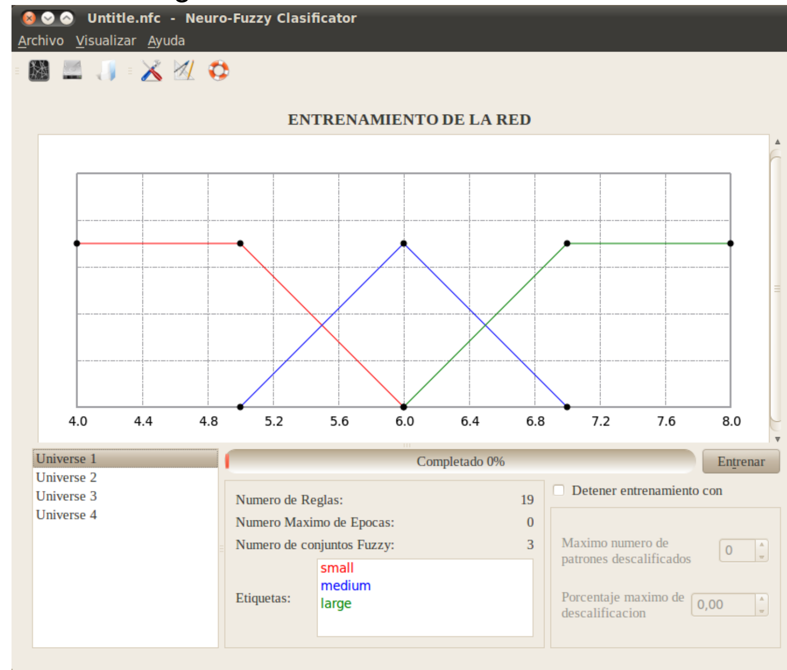
Figura 5.16: Reglas creadas



Fuente: Autores

Cuando tenemos las reglas definitivas podemos pasar al proceso de entrenamiento, entonces nos dirigimos a la ventana de entrenamiento en Visualizar, Entrenar la red o por medio de “Ctrl + T” y aparece una ventana como la de la figura 5.17.

Figura 5.17: Entrenamiento de la red

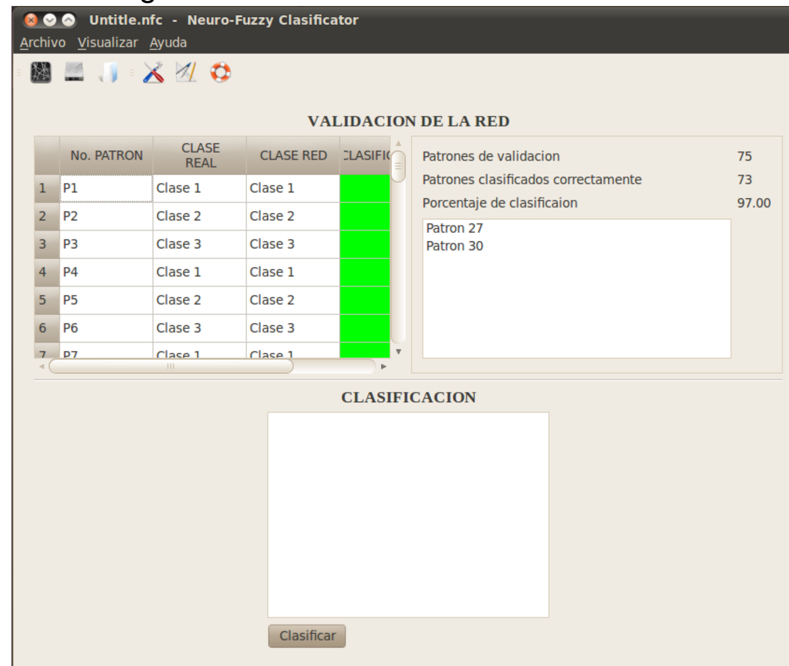


Fuente: Autores

En esta ventana se puede visualizar como van cambiando los conjuntos fuzzy a medida que el entrenamiento va avanzando, de igual forma se puede definir algunos criterios para detener el entrenamiento cuando las condiciones de porcentaje de clasificación o patrones descalificados se cumpla, para ello debemos activarlas chequeando la caja “Detener entrenamiento con”. Para dar inicio al entrenamiento se debe dar click al botón Entrenar y comienza a avanzar la barra de progreso hasta finalizar el entrenamiento. Para evaluar el entrenamiento se debe ir al menú Visualizar, Validar Entrenamiento o “Ctrl + V” automáticamente nos aparece el porcentaje de validación y los patrones que clasifico mal del archivo de validación, como se puede ver en la figura 5.18.

Si la validación del entrenamiento fue lo suficientemente buena se puede pasar al proceso de clasificación, para ello se le da click al botón de clasificar y en seguida aparecerán en la tabla inferior las clases a las que pertenece cada uno de los patrones del archivo de clasificación.

Figura 5.18: Validación del entrenamiento

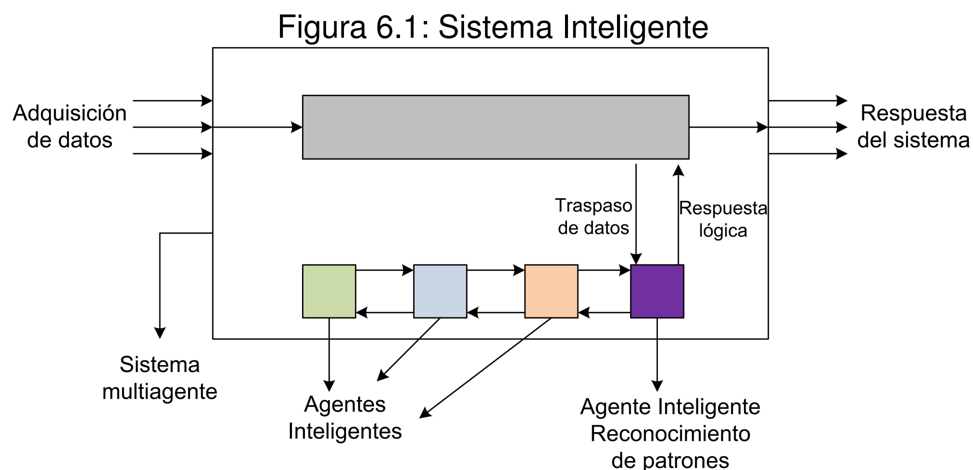


Fuente: Autores

6. CÓMO UTILIZAR LAS LIBRERIAS NEFCLASS-Q?

Las librerías de NEFLASS-Q están destinadas a la creación de una agente inteligente, el cual va a ser parte de un sistema multiagente, donde podrá ser llamado por una aplicación más grande con la función del reconocimiento de patrones de datos, de igual forma tendrá que comunicarse con los demás agentes del sistema, como se puede ver la figura 6.1, de aquí nace la necesidad de que las librerías proporcionen una forma de comunicación con las demás aplicaciones y agentes. Estas librerías tiene un gran potencial y puede llegar a ser aplicadas en diferentes campos como: sistemas de diagnostico de equipos industriales, en la medicina, meteorología, reconocimiento de caracteres etc. De la misma forma como se hizo en el Capitulo 4, se va a utilizar el ejemplo de clasificación de IRIS.DAT para dar un ejemplo de como se debe utilizar correctamente las librerías de NEFCLASS-Q.

Figura 6.1: Sistema Inteligente



Fuente: Autores

Para poder utilizar las libreras es necesario, al igual que la interfaz gráfica de NEFCLASS-Q los archivos de entrenamiento, validación y clasificación, los cuales deben poseer la misma estructura y orden de los datos explicado en el capitulo 4, además es necesario un archivo, el cual debe tener los parámetros necesarios para la creación de la red con la estructura de la figura 6.2.

Figura 6.2: Estructura del archivo de configuración de parámetros

b → Número de variables de entrada

c → Número de clases

d → Número máximo de reglas

$NCON_1$ → Número de conjuntos de la variable 1

$NCON_2$ → Número de conjuntos de la variable 2

.
.
.

$NCON_b$ → Número de conjuntos de la variable b

P → Tipo de pruning

NMR → Número de mejores reglas

EP → Número de epocas

CA_a → Coeficiente de aprendizaje de a

CA_b → Coeficiente de aprendizaje de b

CA_c → Coeficiente de aprendizaje de c

NV_1 → Nombre de la variable 1

NV_2 → Nombre de la variable 2

.
.
.

NV_b → Nombre de la variable b

$NCLA_1$ → Nombre de la clase 1

$NCLA_2$ → Nombre de la clase 2

.
.
.

$NCLA_c$ → Nombre de la clase c

Fuente: Autores

En el caso particular de IRIS.DAT, un archivo de configuración de parámetros es mostrado en la figura 6.3.

Figura 6.3: Archivo de configuración de parámetros de IRIS

```
1 4
2 3
3 50
4 3
5 3
6 3
7 3
8 1
9 20
10 1000
11 0.05
12 0.05
13 0.05
14 Longitud de sepalo
15 Ancho de sepalo
16 Longitud del petalo
17 Ancho del petalo
18 Iris Setosa
19 Iris Versicoulor
20 Iris Virginica
```

Fuente: Autores

en la primera linea de la figura 6.3 se encuentra el número de variables de entrada, que en este caso es 4, en la segunda linea se encuentra el número de clases, que en esta caso es de 3, en la linea 3 se ubica el número máximo de reglas a crear (50), de la linea 4 a la 7 se ubican el número de conjuntos fuzzy que se desean crear para cada una de las variables (para las 4 variables se desea crear 3 conjuntos fuzzy), en la linea 8 se ubica el tipo de pruning, estos son codificados de la siguiente manera:

- El valor 1 representa el pruning simple.
- El valor 2 representa el pruning de las mejores reglas.
- El valor 3 representa el pruning de las mejores reglas por clase.

en este caso el archivo posee pruning simple, en la linea 9 se debe colocar el número de mejores reglas en caso de que se haya escogido el pruning de las mejores reglas o mejores reglas por clase, de lo contrario se ignorara el valor, pero es necesario colocar un valor para que la lectura del archivo de haga de forma correcta. En la linea 10 se encuentra el número de Épocas al que se va a someter el entrenamiento de la red, los siguientes 3 valores de las lineas 11 a 13 se ubican los coeficientes de aprendizaje[1](se utiliza el valor de 0,05 para los tres coeficientes), de las lineas 14 a la 17 se encuentran los nombres de cada una de las variables (Longitud sépalo, ancho sépalo, longitud pétalo, ancho pétalo) y finalmente de las lineas 18 a 20 se ubican los nombres de las clases (Iris Setora, Iris versicolour, Iris Virginica).

[1]La forma como actúa cada coeficiente se encuentra en el capitulo 5, figura 5.11

Para que el agente inteligente realize los principales procesos, se debe crear un archivo en C++, lo primero que debe contener este archivo es la inclusión de las librerías de NEFCLASS-Q, y las estándar de C++ de la siguiente forma:

```
1  #include <iostream>
2  #include "fuzzyset.h"
3  #include "ruleneuron.h"
4  #include "universe.h"
5  #include "files.h"
6  #include "network.h"
7  #include "neurofuzzynet.h"
```

Después de haber incluido las librerías se prepara la función principal de C++ como sigue:

```
1  #include <iostream>
2  #include "fuzzyset.h"
3  #include "ruleneuron.h"
4  #include "universe.h"
5  #include "files.h"
6  #include "network.h"
7  #include "neurofuzzynet.h"
8
9  int main(int argc, char *argv[]){
10
11     return 0;
12 }
```

Y finalmente se prosigue a colocar las funciones que representan los procesos principales de NEFCLASS-Q

```
1  #include <iostream>
2  #include "fuzzyset.h"
3  #include "ruleneuron.h"
4  #include "universe.h"
5  #include "files.h"
6  #include "network.h"
7  #include "neurofuzzynet.h"
8
9  int main(int argc, char *argv[]){
10     NeuroFuzzyNet N;
11     N.initializeLabels();
12     N.readFileParamNet("/home/fabio/Escritorio/red");
13     N.createNeuroFuzzyNet("IRIS1.dat");
14     N.configureLearning();
15     N.createRules();
16     N.training();
17     N.validation("IRIS2.dat");
```

```
18        N.classification("IRIS3.dat");
19        return 0;
20    }
```

En la linea 10 se crea un objeto ***N*** que representa la red Neuro-Fuzzy, con este objeto se puede llamar una serie de funciones, con las que se va a crear los conjuntos fuzzy, crear las reglas, entrenamiento, etc. La primera función que se debe llamar es la de la linea 11, con la que se les asigna las etiquetas según el número de conjuntos fuzzy que el usuario haya definido en el archivo de configuración de parámetros, en la linea 12 se llama la función que lee el archivo de configuración de parámetros, en la que se le debe pasar como argumenta la ruta en la que esta guardado dicho archivo, en la linea 13 se llama la función que crea la estructura básica de la red Neuro-Fuzzy, es decir se van a crear la neuronas de entrada, las neuronas de salida y los conjuntos fuzzy, el argumento que se le debe pasar a esta función es la ruta del archivo de entrenamiento, en la linea 14 se llama la función que configura los parámetros de aprendizaje, es decir, se define el tipo de pruning que se le va aplicar las reglas, el número de mejores reglas si es necesario, el número de épocas y el coeficiente de aprendizaje, en la linea 15 se llama la función que me crea las reglas y hasta aquí se ha definido completamente la estructura de Red Neuro-Fuzzy.

En la linea 16 se llama la función que realiza el entrenamiento a la red de acuerdo a las configuraciones que se hicieron para este proceso, en la linea 17 se llama la función que realiza la validación del entrenamiento, a esta función se le debe pasar como argumento la ruta del archivo de validación y por último en la linea 18 se llama la función que realiza la clasificación para los patrones de datos que contenga el archivo que se le pasa como argumento a esta función.

Cuando se hace correr el programa, automáticamente se genera un archivo de texto en el que se encuentra los conjuntos fuzzy de todas las variables modificados después de realizado el entrenamiento y las reglas creadas por las librerías NEFCLASS-q en el lenguage *fpl* (*fuzzy programming language*), este archivo de salida es mostrado en el Anexo C , el lenguage fpl fue desarrolado por Togai InfraLogic (TIL), especificamente en la implemetación de sistemas basados en lógica fuzzy, este lenguage consiste de 15 objetos diferentes que proporcionan facilidad y flexibilidad a la hora de definir sistemas fuzzy, esto con el fin pueda ser entendido por cualquier persona e igualmente con la intención de que pueda ser comunicado con otras aplicaciones, además de esto, se presentaran los resultados de validación y clasificación.

De esta forma se asegura que el agente inteligente trabaje adecuadamente.

7. PRUEBA DE NEFCLASS-Q CON IRIS.DAT

Para la prueba del programa se utilizó una base de datos conocida como Iris.dat, esta es la más utilizado en el campo del reconocimiento de patrones de datos, esta base de datos consiste de información de tres tipos de flores:

- Iris Setosa
- Iris Versicolour
- Iris Virginica

Las cuales pueden ser clasificados conociendo características como el ancho y la longitud del sépalo y del pétalo, en el archivo se encuentran los datos en el siguiente orden como se puede ver en la figura 7.1:

- Longitud del sépalo en cm.
- Ancho del sépalo en cm.
- Longitud del pétalo en cm
- Ancho del pétalo en cm.

Figura 7.1: Archivo Iris.dat

```
 1 75 4 3
 2 4 8
 3 2 5
 4 1 7
 5 0 3
 6 5.1  3.5  1.4  0.2  1 0 0
 7 7.0  3.2  4.7  1.4  0 1 0
 8 6.3  3.3  6.0  2.5  0 0 1
 9 4.9  3.0  1.4  0.2  1 0 0
10 6.4  3.2  4.5  1.5  0 1 0
11 5.8  2.7  5.1  1.9  0 0 1
12 4.7  3.2  1.3  0.2  1 0 0
13 6.9  3.1  4.9  1.5  0 1 0
14 7.1  3.0  5.9  2.1  0 0 1
15 4.6  3.1  1.5  0.2  1 0 0
16 5.5  2.3  4.0  1.3  0 1 0
17 6.3  2.9  5.6  1.8  0 0 1
18 5.0  3.6  1.4  0.2  1 0 0
19 6.5  2.8  4.6  1.5  0 1 0
```

Fuente: Autores

El archivo contiene 150 patrones, de los cuales cada tipo de flor presenta 50 casos, cada flor tiene codificada un vector de 3 números, para facilitar la lectura de los archivos. La codificación es la siguiente:

- Clase 1 0 0 es Iris Setosa
- Clase 0 1 0 es Iris Versicolour
- Clase 0 0 1 es Iris Virginica.

Para la valoración del programa fue necesario dividir la base de datos en dos archivos llamados IRIS1.dat y IRIS2.dat. donde presentan los casos aleatoriamente a diferencia de IRIS.dat que tiene 75 patrones organizados por clases. Esto es importante porque si el archivo de entrenamiento recibe los datos organizados, no va a presentar buenos resultados. El archivo IRIS1.dat se utilizo para crear las reglas y el entrenamiento, mientras que el archivo IRIS2.dat se utilizo para evaluar el entrenamiento.

Para crear una red es necesario primero definir algunos parámetros que van a caracterizar el entrenamiento, estos parámetros son los siguientes:

- Número de conjuntos fuzzy en cada Universo de discurso.
- Etiquetas de cada uno de los conjuntos.
- Número de Reglas.
- Tipo de Pruning que se le desea aplicar a las reglas.
- Número de Épocas de entrenamiento.
- Ratas de aprendizaje.

Debido a que no hay una forma de decir con cuales parámetros vamos a obtener la mejor clasificación, se decidió variar algunos de ellos y analizar los resultados.

7.1. PRUNING SIMPLE.

Con este pruning se dejan todas las reglas creadas en el proceso de aprendizaje utilizando el algoritmo de aprendizaje en el que se propaga los patrones del archivo de entrenamiento para obtener un número de reglas. Para los resultados de este método se varió el número de conjuntos fuzzy, y el coeficiente de aprendizaje.

7.1.1. Resultados para el método pruning simple.

Tabla 7.1: Tabla resultados pruning simple

Coef. Aprendiz	# Conj	Épocas	Validación	Reglas creadas
0.001	3	1000	97 %	19
0.001	5	2000	96 %	37
0.001	7	4000	86 %	48
0.01	3	1000	97 %	19
0.01	5	2000	96 %	37
0.01	7	4000	86 %	48
0.1	3	1000	96 %	19
0.1	5	2000	94 %	37
0.1	7	4000	88 %	48

La tabla 7.1 muestra los resultados que se obtuvieron con pruning simple variando el número de conjuntos fuzzy por Universo y los coeficientes de aprendizaje. Se puede ver claramente que un coeficiente de aprendizaje muy grande influye de manera negativa el entrenamiento de la red, también podemos ver que si aumentamos el número de conjuntos fuzzy con el pruning simple no se obtienen buenos resultados, en conclusión lo recomendado para utilizar el pruning simple es pocos conjuntos fuzzy por universo y coeficientes de aprendizaje pequeños.

7.2. PRUNING DE LAS MEJORES REGLAS

Este Pruning elimina las reglas que tengan menor activación acumulada, para ello hay que tener definido cuantas reglas se quieren almacenar.

7.2.1. Resultados con pruning de las 15 mejores reglas.

En la tabla 7.2 se muestra los resultados para el pruning de las 15 mejores reglas, se puede ver que aunque hay menos reglas con respecto al pruning simple, los resultados son iguales o mejores en el caso cuando se tiene un número mayor de conjuntos fuzzy, este método se utiliza cuando se quiere el conocimiento quiera ser interpretado más fácilmente por un ser humano.

Tabla 7.2: Resultados pruning 15 mejores reglas

Coef. Aprendiz	# Conj	Épocas	Validación	Reglas creadas	Reglas definitivas
0.001	3	1000	97 %	19	15
0.001	5	2000	96 %	37	15
0.001	7	4000	93 %	48	15
0.01	3	1000	97 %	19	15
0.01	5	2000	96 %	37	15
0.01	7	4000	89 %	48	15
0.1	3	1000	96 %	19	15
0.1	5	2000	96 %	37	15
0.1	7	4000	89 %	48	15

7.2.2. Resultados con pruning de las 10 mejores reglas.

Tabla 7.3: Resultados pruning 10 mejores reglas

Coef. Aprendiz	# Conj	Épocas	Validación	Reglas creadas	Reglas definitivas
0.001	3	1000	96 %	19	10
0.001	5	2000	66 %	37	10
0.001	7	4000	96 %	48	10
0.01	3	1000	96 %	19	10
0.01	5	2000	65 %	37	10
0.01	7	4000	96 %	48	10
0.1	3	1000	94 %	19	10
0.1	5	2000	68 %	37	10
0.1	7	4000	94 %	48	10

En la tablas 7.3 se presentan los resultados de la red con el pruning de las 10 mejores reglas, se puede ver que este método no fue muy bueno para el entrenamiento con 5 conjuntos fuzzy, pero para 3 y 7 conjuntos los resultados si estuvieron bien.

7.3. PRUNING DE LAS MEJORE REGLAS POR CLASE

Este pruning tiene como objetivo dejar las reglas que tenga la mejor activación acumulada por clase, para ello se debe tener definido cuantas reglas por clase se van a dejar en funcionamiento. La activación acumulada por clase es la suma de las activaciones de las reglas para cada respuesta de la neurona de salida.

7.3.1. Resultados para 15 y 10 mejores reglas por clase.

Tabla 7.4: Resultados para las 15 mejores reglas por clase

Coef. Aprendiz	# Conj	Épocas	Validación	Reglas creadas	Reglas definitivas
0.001	3	2000	97 %	19	15
0.001	5	4000	96 %	37	15
0.001	7	8000	97 %	48	15
0.01	3	1000	97 %	19	15
0.01	5	1000	96 %	37	15
0.01	7	4000	97 %	48	15
0.1	3	1000	96 %	19	15
0.1	5	2000	96 %	37	15
0.1	7	4000	96 %	48	15
0.001	3	1000	97 %	19	10
0.001	5	1000	96 %	37	10
0.001	7	4000	97 %	48	10
0.01	3	1000	97 %	19	10
0.01	5	2000	96 %	3	10
0.01	7	4000	97 %	48	10
0.1	3	1000	96 %	19	10
0.1	5	2000	96 %	37	10
0.1	7	4000	94 %	48	10

En la tabla 7.4 se presenta los resultados para el pruning de las 15 mejores reglas por clase, se puede ver que con este método se obtienen mejores resultados en todas las variaciones de conjuntos fuzzy y coeficientes de aprendizaje, por ende este es un método en el que podemos confiar.

8. IMPLEMENTACIÓN DE NEFCLASS-Q AL ANÁLISIS DE ESPECTROS DE VIBRACIONES

De las distintas técnicas aplicables al mantenimiento predictivo, el análisis de vibraciones es la más popular. La razón la encontramos en posibilidad de determinar una gran cantidad de defectos, en una amplia gama de maquinas con una inversión inicial razonable. La vibración es uno de los indicativos más claros del estado de la maquina, bajos niveles de vibración indican que la maquina está en buen estado, cuando los niveles están altos indica que algo anda mal.

El análisis espectral consiste en realizar una transformación de una señal en el tiempo al dominio de frecuencias, donde podemos identificar la vibración característica de cada uno de los componentes o defectos que puede presentar nuestro equipo. Para entrenar y validar NEFCLASS-Q es necesaria una cantidad considerable de espectros que no alcanzan a ser cubiertos con los almacenados en la base de datos del laboratorio y la generación de espectros adicionales implica un proceso dispendioso y costoso, por este motivo se decidió tomar los espectros teóricos típicos, creados en el proyecto “SISTEMA NEURO FUZZY: PERSPECTIVAS DE APLICACIÓN EN LA DETECCIÓN DE FALLAS DE EQUIPOS INDUSTRIALES ROTATIVOS” de 1998 en la Escuela de Ingeniería Mecánica de la Universidad Industrial de Santander. Estos espectros fueron creados en el rango de severidad de vibración de la maquina rotativa del grupo, en este banco de espectros se construyeron ejemplos representativos de cada fallas especificados de la siguiente forma:

- 40 espectros de desalineamiento
- 40 espectros de desbalanceo
- 40 espectros de soltura mecánica
- 40 espectros de resonancia
- 40 espectros sin falla.

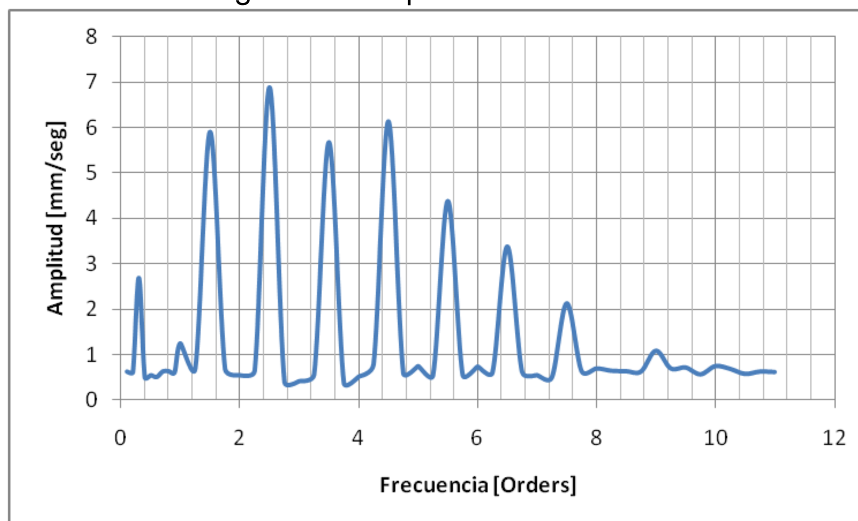
Para la construcción de los archivos fue necesario tomar la amplitud de la vibración de 50 puntos por cada espectro, es decir, dentro del order 0 a 1 se tomaron en cuenta 10 puntos y entre 1 y 11 se tomaron en cuenta 40 puntos separados equitativamente de 0.25, este trabajo fue hecho en el año 1998, un ejemplo de un espectro que será utilizado es el de la figura 8.1 que representa la falla de soltura mecánica.

El patrón de datos que representa el espectro de la figura 8.1 es:

Tabla 8.1: Datos del espectro para el patrón de datos

Frecuencia [Orders]	Amplitud [mm/s]	Frecuencia [Orders]	Amplitud [mm/s]
0.1	0.63	5	0.75
0.2	0.62	5.25	0.55
0.3	2.69	5.5	4.38
0.4	0.5	5.75	0.56
0.5	0.55	6	0.74
0.6	0.51	6.25	0.61
0.7	0.63	6.5	3.38
0.8	0.64	6.75	0.62
0.9	0.61	7	0.55
1.0	1.25	7.25	0.5
1.25	0.7	7.5	2.13
1.5	5.9	7.75	0.64
1.75	0.69	8	0.7
2	0.55	8.25	0.65
2.25	0.65	8.5	0.64
2.5	6.88	8.75	0.63
2.75	0.4	9	1.09
3	0.42	9.25	0.7
3.25	0.57	9.5	0.72
3.5	5.67	9.75	0.57
3.75	0.38	10	0.75
4	0.52	10.25	0.69
4.25	0.79	10.5	0.53
4.5	6.13	10.75	0.63
4.75	0.6	11	0.62

Figura 8.1: Espectro de vibración



Fuente: Autores

La red Neuro-Fuzzy creada por NEFCLASS de 50 entradas que correspondes a los datos de cada patrón de datos como lo muestra la tabla 8.1 fue entrenada con 100 espectros igualmente distribuidos en los tipos de falla, es decir, por cada tipo de fallas habían 20 espectros de entrenamiento. Para la validación se utilizaron 50 espectros igualmente distribuidos en los tipos de falla. Cada uno de los archivos de entrenamiento, validación, y producción tiene que ser presentado a la herramienta en un formato explicado en el cápitulo 4. Se creó una red Neuro Fuzzy con NEFCLASS-Q, en la que se modificaron los parámetros de aprendizaje como el tipo de pruning, el número de conjuntos fuzzy, y el número de mejores reglas y los parámetros de entrenamiento como el coeficiente de aprendizaje y el número de Épocas y presentaron los siguientes resultados:

8.1. ENTRENAMIENTO Y VALIDACIÓN CON PRUNING SIMPLE

Tabla 8.2: Resultados reconocimiento de espectros con pruning simple

Coef. Aprendiz	#Conj	Épocas	Validación	Reglas creadas
0.001	3	4000	96 %	44
0.001	5	12000	90 %	57
0.001	7	15000	74 %	74
0.01	3	2000	98 %	44
0.01	5	8000	90 %	57
0.01	7	10000	76 %	68
0.1	3	1000	96 %	44
0.1	5	5000	90 %	57
0.1	7	5000	80 %	68

8.2. ENTRENAMIENTO Y VALIDACIÓN CON PRUNING DE LAS MEJORES REGLAS

Tabla 8.3: Resultados del reconocimiento de espectros con pruning de las mejores reglas

Coef. Aprend	#Conj	Épocas	Validac	Num.Mej.Reglas
0.01	3	2000	96 %	44
0.01	3	2000	72 %	40
0.01	3	2000	70 %	35
0.01	3	2000	68 %	30
0.01	3	2000	64 %	25
0.01	5	2000	90 %	57
0.01	5	3000	90 %	55
0.01	5	3000	68 %	50
0.01	5	3000	68 %	45
0.01	5	3000	64 %	40
0.01	7	10000	76 %	68
0.01	7	5000	76 %	65
0.01	7	5000	76 %	60
0.01	7	5000	74 %	55
0.01	7	5000	76 %	50

8.3. ENTRENAMIENTO Y VALIDACIÓN CON PRUNING DE LAS MEJORES REGLAS POR CLASE

Tabla 8.4: Resultado reconocimiento de espectros con el pruning mejores reglas por clase

Num.Mej.RegxClase	Coef. Aprend	#Conj	Épocas	Validac	Reglas creadas
8	0.01	3	2000	84 %	28
7	0.01	3	2000	82 %	25
6	0.01	3	2000	80 %	22
5	0.01	3	2000	78 %	19
4	0.01	3	2000	76 %	16
11	0.01	5	5000	82 %	41
10	0.01	5	5000	78 %	38
9	0.01	5	5000	82 %	35
7	0.01	5	5000	80 %	32
5	0.01	5	5000	72 %	21
13	0.1	7	5000	72 %	52
11	0.1	7	5000	72 %	46
9	0.1	7	5000	70 %	38
7	0.1	7	5000	66 %	30
5	0.1	7	5000	64 %	22

De los entrenamientos realizados, el que mejor resultado obtuvo por medio de la validación fue con el pruning simple con 3 conjuntos fuzzy, y un coeficiente de aprendizaje de 0.01. Con esta configuración, la validación obtuvo un resultado del 98 %, clasificando mal 1 espectro, creando las siguientes reglas.

Tabla 8.5: Reglas creadas con los datos de espectros de vibraciones

N°	ANTECEDENTE DE LA REGLA FUZZY	CONCLUSIÓN
1	s,s,s,s,m,s,s,s,s,l,s,m,s,l,s,m,s,m,s,l,s,m,s,m,s,m,s,m,s,m,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
2	s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
3	s,s,s,s,s,s,s,s,s,m,s,m,s,m,s,s,s,l,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
4	s,s,s,s,m,s,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
5	s,s,s,s,s,s,s,s,s,s,s,s,s,l,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
6	s,s,s,s,m,s,s,s,s,l,s,l,s,m,s,m,s,l,s,m,s,m,s,m,s,s,s,m,s,m,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
7	s,s,s,s,m,s,s,s,s,m,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,m,s,s,s,l,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
8	s,s,s,s,s,s,s,s,s,l,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,	Desalineamiento
9	s,s,s,s,m,s,s,s,s,l,s,m,s,l,s,s,s,m,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
10	s,s,s,s,m,s,s,s,s,m,s,m,s,m,s,m,s,m,s,m,s,m,s,s,s,m,s,m,s,m,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
11	s,s,s,s,s,s,s,s,s,m,s,s,s,l,s,s,s,l,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
12	s,s,s,s,s,s,s,s,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,l,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
13	s,s,s,s,s,s,s,s,s,s,s,m,s,m,s,l,s,m,s,l,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
14	s,s,s,s,s,s,s,s,s,l,s,s,s,l,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
15	s,s,s,s,s,s,s,s,s,s,s,l,s,s,s,l,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
16	s,s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Resonancia
17	s,s,s,s,s,s,s,s,s,s,s,m,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
18	s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Resonancia
19	s,s,s,s,s,s,s,s,s,m,s,s,s,l,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
20	s,s,s,s,s,s,s,s,s,l,s,s,s,m,s,s,s,s,s,s,s,m,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
21	s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Resonancia
22	s,s,s,s,s,s,s,s,s,s,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
23	s,s,s,s,m,s,s,s,s,m,s,l,s,m,s,m,s,m,s,m,s,m,s,s,s,m,s,m,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
24	s,m,s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desbalanceo

Tabla 8.7: continuación de las reglas creadas para clasificación de espectros de vibración

REGLA#	ANTECEDENTE DE LA REGLA FUZZY	CONCLUSIÓN
25	s,s,s,s,s,s,s,s,s,s,s,l,s,s,s,l,s,s,s,l,s,s,s,l,s,s,s,l,s,s,s,l,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
26	s,s,s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
27	s,s,s,s,s,s,s,s,s,l,s,l,s,s,s,m,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
28	s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Resonancia
29	s,s,s,s,s,s,s,s,s,m,s,s,s,l,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
30	s,s,s,s,s,s,s,s,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
31	l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Resonancia
32	s,s,s,s,s,s,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
33	s,s,s,s,s,s,s,s,s,l,s,l,s,m,s,m,s,m,s,l,s,m,s,l,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
34	s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Resonancia
35	s,s,s,s,s,s,s,s,s,m,s,s,s,m,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
36	s,s,s,s,m,s,s,s,s,s,s,m,s,s,s,m,s,l,s,m,s,s,s,m,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
37	s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Resonancia
38	s,s,s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desbalanceo
39	s,s,s,s,s,s,s,s,s,m,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento
40	s,s,s,s,s,s,s,s,s,s,s,l,s,s,s,l,s,s,s,l,s,s,s,l,s,s,s,m,s,s,s,m,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Soltura mecánica
41	s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Sin falla
42	s,s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Resonancia
43	s,s,s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desbalanceo
44	s,s,s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,l,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s	Desalineamiento

Como la validación obtuvo buenos resultados queriendo decir que el entrenamiento fue un éxito, se puede pasar al siguiente paso de clasificación.

CONCLUSIONES

- Se cumplieron cabalmente los objetivos propuestos en el plan del trabajo de grado, comprobando el potencial y la exactitud de la herramienta desarrollada NEFCLASS-Q con la base de datos IRIS.DAT y con el reconocimiento de espectros de vibración alcanzando una eficiencia del 98 %.
- Las librerías desarrolladas para el programa quedaron fácilmente adaptables para otro programa, y con las que se podrá crear, entrenar y validar con cortos códigos en c++ por medio del llamado de algunas funciones, esto con el fin de facilitar la creación de los agentes inteligentes.
- El método de pruning de las mejores reglas por clase, es el mejor método para tener una base de conocimiento más pequeño sacrificando un poquito de exactitud.
- Cuando se utiliza un número grande de conjuntos fuzzy en el pruning simple, se generan un gran número de reglas y por ende el resultado de la validación no es bueno.
- El valor adecuado del coeficiente de aprendizaje depende del problema (el dominio de las variables), pero cuando se utiliza un coeficiente de aprendizaje grande el entrenamiento de la red es menos confiable.
- Cuando el número de conjuntos fuzzy es mayor es necesario entrenar la red con un número mayor de épocas,

RECOMENDACIONES

- Se proyecta en futuro próximo implementar las librerías de NEFCLASS-Q, en un sistema automático de diagnostico de los sistemas de bombeo mecánico de crudo, con la ayuda del reconocimiento de forma de los dinagramas de fondo del sistema.

- Se sugiere al Laboratorio de Vibraciones Mecánicas la utilización de la herramienta NEFCLASS-Q, como una ayuda académica, ya que esta herramienta proporcionar el conocimiento al usuario por medio de reglas lingüísticas fácilmente interpretables y que sirva como una guía en el análisis de espectros de vibraciones.

- Para los estudiantes de la Escuela de Ingeniería Mecánica se recomienda entrar al campo de la Inteligencia Artificial y así de esta forma crear versiones posteriores de NEFCLASS-Q.

BIBLIOGRAFÍA

- BRAMER, Max y DEVEDZIC, Vladan. Artificial Intelligence Applications and Innovations. Kluwer Academic Publisher. Touluse,France. 2004.

- COPPIN, Ben. Artificial Intelligence Iluminated. Jones and Bartlett Publishers Inc. London. 2004.

- GUERRERO, Joaquin y RODRIGUEZ,Carlos. Sisteman Neuro-Fuzzy:Prospectivas de aplicación en la detección de fallas en quipos industriales rotativos. Bucaramanga 1998. Trabajo de grado (Ingeniero Mecánico). Universidad Industrial de Santander. Escuela de Ingeniería Mecánica.

- HAGAN, Martin; DEMUTH, Howard y BEALE, Mark. Neural Networks Design. PWS Publishing. 1996.

- HAYKIN, Simon. Neural Networks: A comprehensive foundation, Segunda Edicion, Pretince Hall, 1999.

- KLAWONN, Frank; DETLEFT, Nauck y KRUSE, Rudolf. Generating Rules from Data by Fuzzy ans Neuro-Fuzzy Methods .Technical University of Braunschweig, Department of Compiter Science, Germany.

- KONAR, Amit. Artificial Intelligence and Soft Computing: Behavioral and Cognitive Modeling of the Human Brain. CRC Press. London. 2000.

- KRUSE, Rudolf; NAUCK, Detleft. Learning Methods for Fuzzy Systems. Technical University of Braunschweig, Department of Compiter Science, Germany.

- LUGER, George. Artificial Intelligence: Structures and Strategies for complex problems solving. Fifth edition. Addison Wesley. 2005.

- MELLIN, Patricia et all. Foundations of Fuzzy Logic and Soft Computing. Springer. Cancun, Mexico. 2007.

- NAUCK, Detleft; KLAWONN, Frank y KRUSE, Rudolf. Fuzzy Set, Fuzzy Controller, and Neural Networks. Technical University of Braunschweig, Department of Compiter Science, Germany.

- —-, —-. Neuro-fuzzy Classification Initialized by Fuzzy Clustering. Technical University of Braunschweig, Department of Compiter Science, Germany.

- —-, KRUSE, Rudolf. A Fuzzy Neural Network Larning Fuzzy Control Rules and Membership Function by Fuzzy Error Backpropagation. Technical University of Braunschweig, Department of Compiter Science, Germany.

- —-,—-. NEFCLASS- A Neuro-Fuzzy Aproach for the clasification of data. Technical University of Braunschweig, Department of Compiter Science, Germany.
- —-;—-. Choosing Appropriate Neuro-Fuzzy Model. Technical University of Braunschweig, Department of Compiter Science, Germany.
- NAUCK, Detleft. A Fuzzy perceptron as a generic Model for Neuro-Fuzzy Approaches. Technical University of Braunschweig, Department of Compiter Science, Germany.
- —-. Beyond Neuro-Fuzzy: Perspectives and Directions. Technical University of Braunschweig, Department of Compiter Science, Germany.
- —-, NAUCK, Ulrike y KRUSE, Rudolf. Generating Classification Rules with the Neuro-Fuzzy System NEFCLASS. Technical University of Braunschweig, Department of Compiter Science, Germany.
- NEGNEVITSKY, Michael. Artificial Intellegence: A Guide to Intellegent Sistems. Second Edition. Addison Wesley. 2005.
- PARTRIDGE, Derek. Artificial Intelligence and Software Engineering: Understanding the Promise of the Future. Intellected Ltda. New York.1998.
- PRIEST, Graham. An introduction Non-Classicla Logic.
- VILLASANA, Minaya, Presentación: Introducción a las Redes Neuronales. Segunda Edicion. Universyty of melbourne and University of St Andrews. Cambridge University Press. 2008.Lógica fuzzy

A. LOGICA FUZZY

Esencia de la lógica fuzzy

La lógica fuzzy como su nombre lo indica, es la ciencia que subyace en el razonamiento aproximado en vez del exacto, la importancia de esta se deriva de el razonamiento humano y especialmente del sentido comun del razonamiento. En este sentido, la lógica Fuzzy (LF) es la encargada de establecer los principios formales del razonamiento "aproximado" o "incierto", donde el razonamiento "preciso" es mirado como un caso "particular" (caso límite o extremo). Para el caso de la Lógica Fuzzy lo primordial es tratar de asemejarse a la incertidumbre e imprecisión que puede manejar el ser humano al momento de efectuar un razonamiento bajo parámetros que no son claros o presentan ambigüedad para un programa clásico de programación . Esta habilidad innata en el ser humano depende en esencia , del conocimiento acumulado (almacenado) que es inexacto, incompleto y en muchas ocasiones no es totalmente fidedigno y de la valoración semántica que se le asigna a determinada situación.

Es de interés notar que a pesar de su omnipresencia, el razonamiento aproximado está fuera de los alcances de la lógica clásica, porque este razonamiento está profundamente afianzado a la logica fuzzy. Existen dos razones principales por las cuales los sistemas lógicos clásicos no pueden arreglárselas con problemas de este tipo. La primera es que ellos no proporcionan medios que permitan representar el significado de las proporciones expresadas en lenguaje natural, cuando el significado es impreciso; la segunda es que , aun en aquellos casos en los cuales el significado de las proporciones pueda ser representado simbólicamente en un lenguaje, como por ejemplo en una red semántica, existe un mecanismo para realizar las inferencias.

Como puede ser visto, la lógica fuzzy canaliza estos problemas de la siguiente manera. Primero, el significado de una proposición lexicamente imprecisa, es representado como una restricción elástica (flexible) en una variable; y segundo, la respuesta a una pregunta es deducida a través de la interrelación de restricciones elásticas.

Algunas características escenciales de la lógica fuzzy se relacionan en los siguientes estamentos:

- El razonamiento exacto es visto como un caso límite del razonamiento aproxiamdo.
- Todas las cosas son graduadas.
- Cualquier sistema lógico puede ser fuzzificado.

- El conocimiento es interpretado como una colección elástica de restricciones fuzzy sobre una colección de variables.
- La inferencia es vista como un proceso de propagación de restricciones elásticas .

La inferencia es vista como un proceso de propagación de restricciones elásticas. Durante los años anteriores, la lógica fuzzy ha encontrado numerosas aplicaciones en muy variados campos, desde las finanzas hasta la ingeniería de terremotos, pasando por aplicaciones como medicina, biología, visionica, procesamiento del lenguaje natural, sistemas basados en el conocimiento, etc. Lo verdaderamente sorprendente es que la aplicación más importante y visible, actualmente se encuentra en un área no contemplada cuando la lógica fuzzy fue concebida, dicha área corresponde al "*control de procesos basado en logica fuzzy*".

La lógica fuzzy tiene sus antecedentes en la lógica multivaluada (n-valuada, donde $n \geq 2$) de LUKASIEWICZ, quien a principios de los años 30 la propuso. El término fuzzy (borroso, difuso, nebuloso) es introducido por LOTFI A. ZADEH en 1962, en la Universidad de California en Berkeley, en un trabajo que vincula la teoría de circuitos eléctricos con la de sistemas. La noción de *conjunto fuzzy* aparece por primera vez en 1964 en un trabajo de L. A. ZADEH. Este trabajo sería publicado un año más tarde en la revista "Information and Control" (ZADEH, 1965) marcando el nacimiento de una nueva teoría (o tecnología) conocida genéricamente como "Teoría de Conjuntos Fuzzy", y que coloquialmente hablando, viene a prestar fundamento teórico a la descripción de aquellas propiedades en las que la transición del ser al no ser y del se cumple al no se cumple, es gradual y no brusca. El trabajo de ZADEH en 1965 sugería aplicaciones a sistemas humanísticos y para nada se vislumbraba su aplicación al control de procesos industriales; desde entonces se generó un gran desarrollo teórico y de aplicaciones a una gran variedad de áreas.

Resumiendo se podría decir que la lógica fuzzy puede entenderse como una herramienta útil para construir modelos de razonamiento humano que reflejan el carácter impreciso (vago) y cualitativo que este tiene. El enorme interés levantado por la lógica fuzzy estriba en la posibilidad de manejar problemas demasiado complejos, o muy mal definidos, como para admitir un tratamiento por métodos tradicionales.

Diferencias entre lógica fuzzy y crisp

- Veracidad: En los sistemas clásicos, solo hay dos posibles valores de verdad, verdadero o falso. En la lógica fuzzy los valores de verdad pueden ser un subconjunto fuzzy de algun conjunto parcialmente ordenado, usualmente este valor se asume que esta en el intervalo [0,1] de un subconjunto fuzzy o simplemente un punto en este intervalo. Los llamados valores de verdad lingüísticos expresados como verdadero, muy verdadero, no muy verdadero, etc. son interpretados como etiquetas de subconjuntos fuzzy.

- Predicados: En los sistemas clásicos, los predicados son concretos, por ejemplo mortal, liso, más grande que. En la lógica fuzzy los predicados no son concretos, por ejemplo mal, pronto, veloz, mucho más grande que. Debe tenerse en cuenta que la mayoría de los predicados en el lenguaje natural no son concretos.

- Modificadores de predicado: En los istemas clásicos es usado el modificador de predicado de negacion *not*. En la logica fuzz hay una variedad de modificadores de predicado como por ejemplo muy, mas o menos, bastante, extremadamente.

  Los modificadores de predicado juegan un papel importante en la generación de valores de una variable lingüística, por ejemplo muy joven, no muy joven, mas o menos joven, etc.

- Cuantificadores: En la lógica clásica hay dos cuantificadores: universal y existencial. La lógica fuzzy tiene además de estas una amplia variedad de cuantificadores fuzzy, por ejemplo poco, varios, usualmente, la mayoría, siempre, frecuentemente, etc. En la lógica fuzzy, un cuantificador fuzzy es interpretado como un número fuzzy o una proporción fuzzy.

- Probabilidades: En los sistemas de logica clasica, es un valor númerico o intervalo númerico. En logica fuzzy, se tiene la opción adicionalmente de emplear la lingüística o generalmente las probabilidades fuzzy, por ejemplo probable, no probable, muy pro- bable, altamente probable. Las probabilidades fuzzy pueden ser interpretadas como números fuzzy, los cuales deben ser manipulados a través de la aritmética fuzzy.

  Adicionalmente a las probabilidades fuzzy, la lógica fuzzy hace posible el manejo de los eventos fuzzy. Un ejemplo de un evento fuzzy es mañana será un dían cálido, donde cálido es predicado fuzzy.

- Posibilidades: En contraste al modelo de lógica fuzzy, el concepto de posibilidad en lógica fuzzy es graduado en vez de concreto. Sin embargo como en el caso de las probabilidades, las posibilidades pueden ser tratadas como varibles lingüísticas con valores como posible, poco posible, altamente posible, etc. Estos valores pueden ser interpretados como etiquetas de subconjuntos fuzzy de la linea real. Uno de los conceptos que juega un papel importante en la lógica fuzzy es la distribución de posibilidades.

Significado y representación del conocimiento

El conocimiento puede ser visto como una colección de proposiciones.
Mary es joven.
Pat es mas alta que Mary.

La mayoría de los suecos son rubios.
Los tomates son más rojos si ellos estan maduros.
Usualmente la alta calidad va acompañada de un alto precio.
La mayoría de los hombres altos no son muy ágiles.

En estas oraciones significado y representación estan relacionadas entre sí. En la lógica fuzzy significado y representación estan basados en el concepto de *test-score*. Una idea fundamental del concepto *test-score* es que una proposición en un lenguaje natural puede ser vista como una colección de restricciones fuzzy elásticas, por ejemplo la proposición de *Mary es alta* representa una restricción elástica sobre la altura de Mary. Similarmente la proposición *Jean es rubia* representa una restricción elástica sobre el color del cabello de Jean y la proposición *la mayoría de los hombres altos no son muy ágiles* representa una restricción elástica sobre la proporción de los hombre que no son muy agiles.

En terminos más concretos, la representación y el significado envuelve los siguientes pasos usando el *test-score*:

1. Identificación de las variables $X_1, X_2, X_3, ..., X_n$, en la que los valores son restringidos por la proposición. Usualmente estas variables estan implicitas en vez de explícitas en la proposición.

2. Identificación de las variables $C_1, C_2, C_3, ..., C_n$ las cuales son introducidas por la proposición.

3. Caracterización de cada restricción C_i por la descripción del procedimiento de prueba que asocia C_i con un *test-score* τ que representa el grado con el cual C_i es satisfecho. Usualmente τ_i es expresado como un número en el intervalo $[0, 1]$, sin embargo un *test-score* puede ser una probabilidad dirtribución deo posibilidad en un intervalo.

4. La agregación de un *test-score* parcial. Los cuales son representados como un vector de *test-score* $\tau_i, ..., \tau_k$. En la mayoria de los casos $K = 1$ entonces el *test-score* sería un escalar.

Cuantificación de la ambigüedad

La mayoría de los lenguajes naturales contienen ambigüedad y multiplicidad de significados, que están determinados por factores culturales y que pueden cambiar de forma brusca inclusive en comunidades vecinas. Los propósitos de los adjetivos, especialmente, no son claramente específicos, ellos son ambiguos en términos de la amplitud de su significado. Por ejemplo, si nosotros decimos “Persona alta”, no podemos determinar quién es más alto o quien es menos alto. La ambigüedad (imprecisión, incertidumbre) de “Persona vieja” proviene del adjetivo “viejo”. Las palabras son usualmente cualitativas, pero no persigue que “Alto” o “Viejo” sean percibida en conexión con cantidades asociadas a la altura o la edad.

Fundamento de la teoría de conjuntos fuzzy

Definición de conjuntos booleanos

La imprecisión o ambigüedad está relacionada con la incoherencia entre nuestro entendimiento clásico de los fenómenos y su existencia actual en el mundo real. ¿Cómo puede ser esto? La respuesta recae en las herramientas que usamos para construir mapas entre el mundo real y nuestros modelos. Esas herramientas están basadas en una antigua y penetrante visión del mundo, formalizadas desde los tiempos de ARISTOTELES y refinadas por pensadores tales como GEORGE BODE, GEORGE CANTOR, DAVID HILBERT y BERTRAND RUSELL. Los procesos de razonamiento bajo esta visión, manejados por la lógica booleana (bi-valuada), miran el mundo en términos de categorías bien estructuradas (definidas). Cada categoría (o conjunto) tiene una muy bien definida lista de pertenencia. Un ítem es por lo tanto es miembro de un conjunto o no lo es, no dando lugar a categorías intermedias en cuanto a la pertenencia.

Si se considera un determinado universo de discurso, por ejemplo en conjunto de los números enteros N, generalmente la descripción de los datos se da por la definición de subconjuntos del universo dado. La característica menor que 10 en relación a $\mathbb{N}$ puede ser caracterizada por el conjunto.

$$A = 1, 2, 3, 4, 5, 6, 7, 8, 9 \subseteq \mathbb{N}$$

Igualmente se puede entender menores que 10 como un predicado, por ejemplo, considerando un entero $n \in N$ *menores que*$10(n)$, es verdad sí n es un número de 1 a 9.

$$A = n \in \mathbb{N} | menores\, que\, 10(n)$$

Un tercer tipo de representación de los datos del universo de discurso que son menores que 10, es la definición de la respectiva función característica:

$$\mathbb{I}_A : \mathbb{N} \Rightarrow 0, 1$$

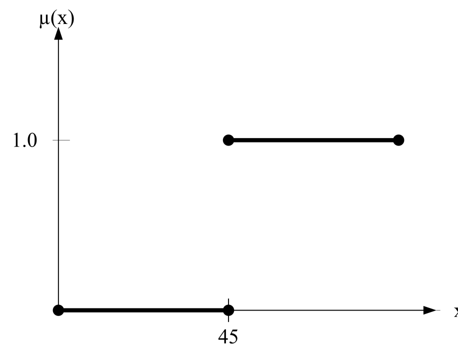
$$\mathbb{I}_A(n) = \begin{cases} 1 & sí\, menores\, que\, 10 \\ 0 & en\, otro\, caso \end{cases}$$

da valor de 1 para cada elemento del universo que pertenece a A y 0 para cada elemento que no pertenece.

Este concepto de conjunto concreto puede ser usado para caracterizar, por ejemplo, “los mayores de 45 años” en el universo de los hombre que viven en Colombia. La función o caracteristica de estos conjuntos es como se muestra en la figura A.1.

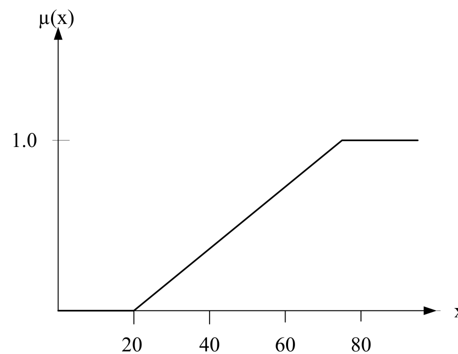
El problema se complica cuando se quiere caracterizar el término “viejo”, el uso de algunas funciones características resulta insatisfactori, ya que habría una edad x donde la persona es considerada vieja, y una edad de $x - 1$ donde ya no se puede considerar la persona como *viejo*. Una posible solución es generalizar la definición de la función característica de manera que el rango de la función característica sea el intervalo de $[0, 1]$.

Figura A.1: Conjunto Concreto



Fuente: Autores

Figura A.2: Conjunto Fuzzy



Fuente: Autores

Definición de conjunto fuzzy:

Un conjunto fuzzy μ de X es una función que asigna a los elementos del universo de discurso un valor dentro de un intervalo. $\mu : X \Rightarrow [0,1]$, donde $F(X)$ representa el conjunto de todos los conjuntos fuzzy de X.

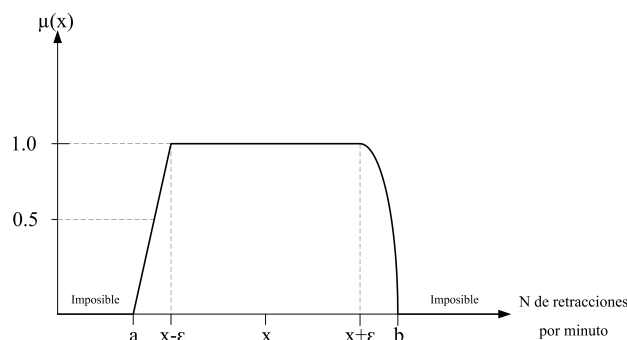
El valor $\mu(x)$ representa el grado de pertenencia del elemento x al conunto fuzzy μ.La figura A.2 muestra una función de pertenencia que describe el termino lingüístico o viejo. En este ejemplo una edad de 70 años se considera que pertenece al concepto viejo con una certeza de 1, mientras que una edad de 20 años no pertenece al conjunto, y una edad de 45 tiene una pertenencia de $0{,}5$ a este conjunto. El uso de los conjuntos fuzzy es representar conceptos vagos y con frecuencia es hecho de manera intuitiva, porque en muchas ocasiones no existe un modelo que intérprete de manera clara el grado de pertenencia.

Por ejemplo sí x es el número de revoluciones por minuto del disco duro de un computador, obviamente esta información es afectada con incertidumbre, por la tanto es más real usar datos lingüísticos.

El número de rotaciones por minuto es cercano a x

El intervalo$[x- \in, x+ \in]$ puede ser usado para modelar, pero esto causa el problema mencinado de contar con limites concretos de $x- \in$ y $x+ \in$. Sí los datos estadísticos no estan disponibles, entonces los métodos de probabilidades no pueden ser usados, los conjuntos fuzzy son una posible solución, donde los expertos pueden especificarlos de una menera directa. En este ejemplo el experto escoge el conjunto fuzzy de la figura A.3. Los valores de x menores que a o más grandes que b son considerados como imposible, mientras que los valores que estan entre x– $\in$ y $x+ \in$ ocurren con certeza. Para el resto de las áreas el experto escoge un incremento monótono y una pendiente negativa de la función respectivamente.

Figura A.3: Conjunto fuzzy que caracteriza el número de rotaciones por minuto de un disco duro



Fuente: Autores

Variable lingüística o variable fuzzy

Cuando se intenta modelar la realidad, a cada fenómeno o hecho se le asocia un nombre de variable en el modelo, para posteriormente ser descrita en términos de su espacio fuzzy, y es a esta variable la que se denomina “Variable Lingüística” por ejemplo: Temperatura, Presión, Edad, Estatura, etc.

Términos lingüísticos

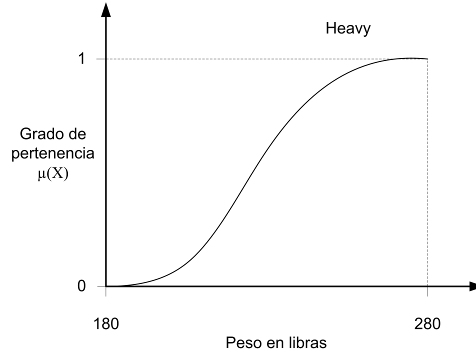
Si tomamos la variable lingüística “Temperatura”, es evidente que nosotros le asociamos múltiples atributos (categorías, etiquetas) semánticos para describirla en todo su espacio fuzzy; en este caso la temperatura podría ser: *caliente, tibia, fresca, fria*. Es a cada uno de estos atributos, lo que se denomina “Términos Lingüísticos” y como se mostro anteriormente, cada uno de ellos puede ser representado a través de un conjunto fuzzy.

Dominio de un conjunto fuzzy

El total de valores permisibles para un término lingüístico asociado a una variable se denomina el dominio del conjunto fuzzy asociado al término lingüístico. El dominio

es un conjunto de números reales, incrementándose monotonicamente de izquierda a derecha. Los valores pueden ser tanto positivos como negativos. Se selecciona el dominio para representar completamente el rango de valores del conjunto fuzzy dentro del contexto de su modelo. Como un ejemplo, el conjunto fuzzy *pesados* (para hombres latinos) tiene un dominio que va de 180 Lbs a 280 Lbs como se ve en la figura A.4.

Figura A.4: Conjunto fuzzy Heavy



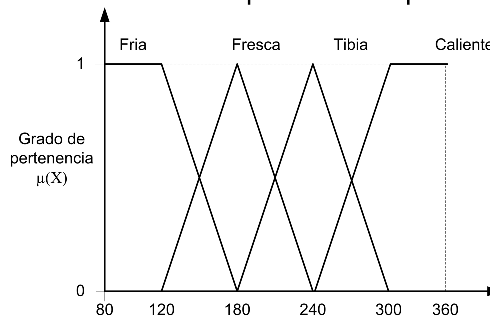
Fuente: Autores

Nótese que el dominio tiene tanto un límite superior como un límite inferior, aunque en realidad, el alcance de los valores asociados con el concepto fuzzy puede ser de extremo final abierto. En el conjunto fuzzy heavy, el peso superior posible puede llegar a ser 280 Lbs. Pero puesto que el conjunto fuzzy alcanza el grado de pertenencia a uno (1.0) en 280 Lbs, se termina el dominio en este punto, dado que cualquier peso por encima de este valor es definitivamente heavy.

Universo de discurso

Una variable en un modelo, como anteriormente se mostro, es a menudo descrita en términos

Figura A.5: Universo de discurso para la temperatura de una turbina

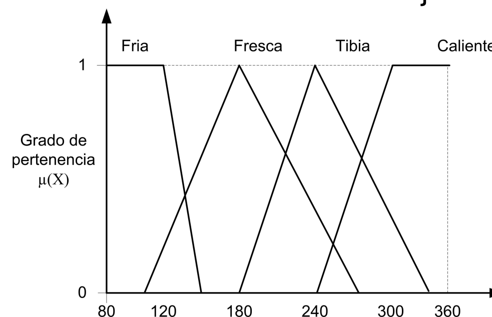


Fuente: Autores

de su espacio fuzzy. Este espacio esta generalmente compuesto de múltiples conjuntos fuzzy (términos o atributos lingüísticos) traslapados, donde cada conjunto fuzzy describe una partición semántica de la variable.

La figura A.5, ilustra este concepto. La variable *temperatura* esta subdividida en cuatro conjuntos fuzzy: *fria*, *fresca*, *tibia* y *caliente*. El espacio total del problema, desde el valor permisible más pequeño hasta el valor permisible mas grande, se denomina "Universo de discurso o de raciocinio". El universo de discurso para la variable *temperatura* (Figura A.6), es de 80ºC a 360ºC, mientras que el dominio del conjunto fuzzy *tibia* es de 180ºC a 300ºC. Los conjuntos fuzzy que describen el universo de discurso no necesariamente son simétricos pero ellos siempre deberían traslaparse en algún grado o porcentaje.

Figura A.6: Universo de discurso con conjuntos asimétricos



Fuente: Autores

Representación de los conjuntos fuzzy

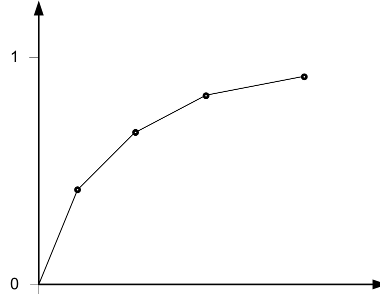
Hay varias posibilidades de representar una función de pertenencia característica de un conjunto fuzzy. Sí el universo X tiene un número finito de elementos, un conjunto fuzzy μ de X será definido por cada elemento x que pertenece a X y su grado de pertenencia $\mu(x)$, sí el número de elementos es muy grande, puede ser mejor definido por una función que use parámetros que se adapten al modelo del problema. Sí $X =$ la expresión lingüística grande puede ser interpretada por una función lineal como:

$$\mu_{a,b}(x) = \begin{cases} 0 & x \leq a \\ 1 & x \geq b \\ \frac{x-a}{b-a} & a < x < b \end{cases} \quad \text{donde se asume que } a \leq b$$

la función mostrada en la figura A.7 es una interpretación alternativa, usando las coordenadas de los puntos definidos como parámetros.

Una función de incremento exponencial

Figura A.7: Conjunto fuzzy definido por puntos



Fuente: Autores

$$\mu_{a,b}(x) = \begin{cases} 1 - e^{-a(x-b)} & b \leq x \\ 0 & b > x \end{cases} \text{ con } a > 0 \text{ y } b \in \mathbb{R}$$

Para interpretar expresiones lingüísticas, se usan los llamados *números fuzzy*, que son simples triángulos simétricos

$$\mu_{m,d}(x) = \begin{cases} 1 - |\frac{m-x}{d}| & m - d \leq x \leq m + d \\ 0 & x < m - d \vee x > m + d \end{cases} \text{ con } d > 0$$

O la funcón Gaussiana

$$\mu_{a,b}(x) = \exp(-a(x-b)^2) \ con \, a > 0, b \in \mathbb{R}$$

La expresión aproximadamente entre a y b puede ser caracterizada por la función trapezoidal

$$\mu_{a,b,c,d}(x) = \begin{cases} \frac{x-a}{b-a} & a \leq x \leq b \\ 1 & b \leq x \leq c \\ \frac{x-d}{c-d} & c \leq x \leq d \\ 0 & x < a \vee x < d \end{cases}$$

la representación de los conjuntos fuzzy usando la caracterización de la función de pertenencia es llamada representación vertical, considerando la representación gráfica de la función. Con frecuencia un experto define un conjunto fuzzy, especificando para todos los grados de pertenencias α de un subconjunto escogido, esos elementos de X que tienen al menos el grado de pertenencia α. Esta es la representación horizontal de un conjunto fuzzy, usando$\alpha - cortes$.

Definicion: si $\mu \in F(x)$ y $\alpha \in [0,1]$ el conjunto

$$[\mu]_\alpha = x \in X | \mu(x) \geq \alpha \text{ es llamado } \alpha - corte \text{ de } \mu.$$

Operaciones sobre conjuntos fuzzy

Los operadores básicos sobre conjuntos como unión, intersección y complemento también pueden ser definidos para conjuntos fuzzy. Es esperado que la intersección $\mu \cap \mu'$ de dos conjuntos fuzzy $\mu, \mu' \in F(x)$ puede ser calculado elemento por elemento, existe una función $\sqcap : [0,1]^2 \to [0,1]$ tal que $(\mu \cap \mu')(x) = \sqcap(\mu(x), \mu'(x))$. Para aceptar $\sqcap$ como un operador de intersección, tiene que cumplir con algunos axiomas, los cuales manejan la definición de la *t-norma*.

Definición: Una función $\top : [0,1]^2 \to [0,1]$ es llamada *t-norma* si cumple las siguientes condiciones:

$$\top(a,1) = a$$

$$a \leq b \Rightarrow \top(a,c) \leq \top(b,c)$$

$$\top(a,b) = \top(b,a)$$

$$\top(a,\top(b,c)) = \top(\top(a,b),c)$$

Obviamente $\top$ no es decreciente en ambos argumentos y $\top(a,0) = 0$. Las *t-normas* más utilizadas son:

$$\top_{\text{mín}}(a,b) = \text{mín}(a,b)$$

$$\top_{LUKA}(a,b) = \text{máx}(0, a+b-1)$$

$$\top_{prod}(a,b) = a{\cdot}b$$

Definición: Una función $\perp : [0,1]^2 \to [0,1]$ es llamada *t-conorma*, sí $\perp$ es conmutativa, asociativa, no decreciente en ambos argumentos y el 0 es su unidad. Algunos ejemplos de $t-conormas$ son:

$$\perp_{\text{mín}}(a,b) = \text{máx}(a,b)$$

$$\perp_{LUKA}(a,b) = \text{mín}(a+b, 1)$$

$$\perp_{prod}(a,b) = a+b-ab$$

las *t-norma* y las *t-conorma* inducen conexiones para conjuntos fuzzy como:

$$(\mu \cap \top\mu')(x) = \top(\mu(x), \mu'(x))$$

$$(\mu \cup \perp\mu')(x) = \perp(\mu(x), \mu'(x))$$

Además de las operaciones entre conjuntos teóricos, es posible extender la asignación de la forma $\phi : X \rightarrow Y$ a la asignación $\hat{\phi} : F(X)^n \rightarrow F(Y)$. La transición de la función ϕ a su extensión $\hat{\phi}$ es llamada principio de extensión.

Definición: Si $\phi : (X)^n \rightarrow Y$ es una asignación. La extensión de ϕ es definida por:

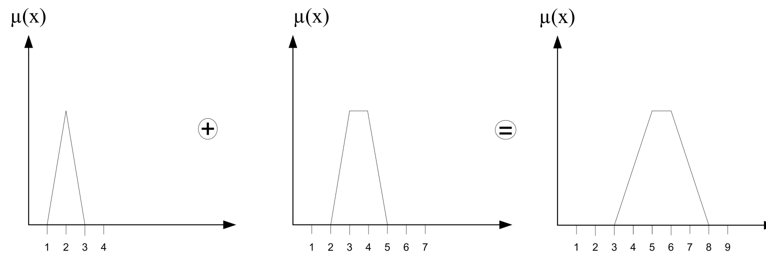
$$\hat{\phi} : F(X)^n \rightarrow F(Y) \text{ con}$$

$$\hat{\phi}(\mu_1, ..., \mu_n)(y) = \sup\{\text{mín}\{\mu_1(x_1), ..., \mu_n(x_n)\} \,|\, (x_1, ..., x_n) \in x^n \wedge y = \phi(x_1, ..., x_n)\}$$

Con la ayuda del principio de extensión podemos obtener la adición de dos conjuntos fuzzy arbitrarios $\mu, \mu' \in F(\mathbb{R})$:

$$(\mu + \mu')(t) = \sup\{\text{mín}\{\mu(x_1), \mu'(x_2)\} \mid x_1, x_2 \in \mathbb{R} \wedge x_1 + x_2 = t\}$$

Figura A.8: Adición de conjuntos fuzzy



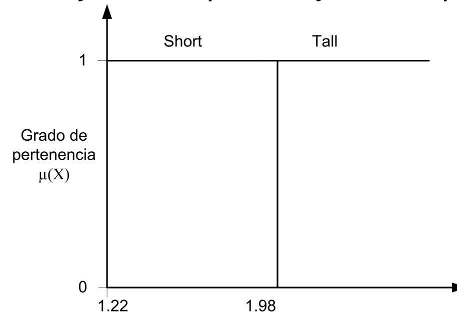
Fuente: Autores

La figura A.8 muestra, la adición de dos conjuntos fuzzy, si como en este caso los conjuntos son convexos, es fácil operar sobre ellos. En general las operaciones sobre conjuntos fuzzy son mucho más complicadas, especialmente si a es usada la representaciín vertical. Así de este modo es preferible la representación horizontal, y operar sobre los α – *cortes*.

Ley de la no contradicción

Una intutitiva , y para algunos, una obvia observación, es que la intersección de un conjuntos_1, con su complemento $\sim s_1$, es un conjunto vacio. Lo anterior se denomina la *Ley de la no contradicción* y tiene sus raices en el "Organon" de ARISTOTELES. Lo anterior se puede ver claramente en la figura A.9, donde la pertenencia para *bajo* (Esto es $\sim alto$) y *alto* se muestra. Es de observar que no existe traslape (`overlap`) entre las dos regiones.

Figura A.9: Conjunto crisp short y su complemento tall

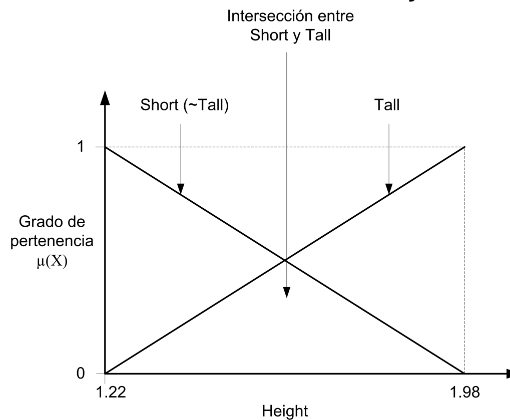


Fuente: Autores

¿Existe la misma categoria de particionamiento en el complemento de conjuntos fuzzy? La respuesta es no. Puesto que un complemento fuzzy indica el grado con el cual un valor de dominio es compatible con el concepto del complemento fuzzy, se podría esperar y de hecho encontrar que existe una cierta cantidad de ambigüedad cuando intentamos particionar un conjunto fuzzy. Esto es, en alguna región del conjunto, existen valores del dominio en el conjunto fuzzy base y en su complemento.

Para ver lo anterior, consideramos la intersección de los conjuntos fuzzy, tall y short ($\sim tall$), iluastrada en la figura A.10. En lugar de dos regiones dislocadas y separadas claramente, existe un traslape parcial entre las dos regiones fuzzy, creando un area (sombreada) donde existen valores que son considerados tanto tall como short ($\sim\ tall$). Lo anterior corresponde a nuestra experiencia diaria donde existe algunos individuos que no son absolutamente tall o short; ellos, desde nuestra apreciación poseen pertenencia a ambos conceptos. En nuestro lenguaje de modelamiento fuzzy, se podría construir una regla perfectamente aceptable, tal como: *Si estatura es tall y estatura es short($\sim tall$) entonces peso es middle.*

Figura A.10: Intersección de conjuntos fuzzy



Fuente: Autores

Esta categoría de modelamiento de la realidad tiene intrínsecas características

de lógica fuzzy. En representaciones booleanas, tal particionamiento no podría existir puesto que violaría la ley de *no contradicción*: “ alguna cosa no puede ser *alta* y *alta* al mismo tiempo”.

Semántica de los conjuntos fuzzy

La existencia de diferentes interpretaciones de probabilidades, y la dificultad de adquirir probabilidades subjetivas, muestra la importancia de aclarar la semántica de los conceptos teóricos. Se pueden considerar dos enfoques semánticos, el primero considera los conjuntos fuzzy como información comprimida e imprecisa, y el segundo enfoque ve a los conjuntos fuzzy como puntos concretos en un ambiente vago.

El primer enfoque usa los conjuntos fuzzy como representación condensada de la imprecisión, generalmente información inconsistente sobre algo desconocido, pero que tiene un valor. Esto es una representación epistémica de conjuntos fuzzy y físicamente una medida del error es asumida en estos casos.

Lo que se asume en este modelo

es la imprecisión de los datos causados por la existencia de un valor dado a la información de un sistema C.

Definición: Sí $(C, 2^c, P_c)$ es un espacio finito de probabilidades, X es un universo arbitrario que no esta vacío, y $\Gamma : C \to 2^X$ un conjunto de valores asignados. El par (P_C, Γ) es llamado *conjunto aleatorio*.

Por ejemplo, se ha entrevistado a 4 personas, en donde se les ha preguntado que altura debe tener una mujer para considerarla alta? sus respuestas son utilizadas para representar el valor lingüístico *alta* de una forma condensada. La primera persona considera que una mujer es alta después de $160\,cm$ de altura, la segunda persona las considera alta después de $165\,cm$, la tercera persona $170\,cm$, y la cuarta persona $180\,cm$. Cada una de las personas forman un contexto, donde se tiene $C = \{c_1, c_2, c_3, c_4\}$, debido a que no hay preferencias se considera $P_c(\{c_i\}) = 0{,}25$ $i = 1, 2, 3, 4$, la asignación $\Gamma : C \to 2^{\mathbb{R}}$ es determinada por $c_1 \to [160, \propto)$,$c_2 \to [165, \propto)$,$c_3 \to [170, \propto)$,$c_4 \to [180, \propto)$.

La pregunta que surge es, puede una mujer de $168\,cm$ de altura llamarse alta? La respuesta a esta pregunta es *“si”* para los contextos c_1 y c_2 y *“no”* en los contextos c_3 y c_4. Entonces la compresión de la información para cada altura puede ser, escribiendo para cada valor $x \in \mathbb{R}$ el peso:

$$\frac{|\,\{c \in C \mid x \in \Gamma(c)\}\,|}{|C|} = \sum_{c \in C : x \in \Gamma(c)} P_C(\{C\})$$

Definición: sí $\top$ es una *t-norma* una asignación $E : X \times X \to [0, 1]$ es llamada *relación de igualdad* con respecto a $\top$ si las siguientes propiedades se cumplen:

- $E(x,x) = 1$ Reflexividad
- $E(x,x') = E(x',x)$ Simetría
- $\top(E(x,x'), E(x',x'')) \leq E(x,x'')$ Transitividad

El valor $E(x,x')$ es interpretado como el grado de aceptación, en el que x y x' son considerados iguales o no distinguibles respectivamente, el ejemplo más simple para una relación de igualdad es la usual igualdad sobre x

$$E(x,x') = \begin{cases} 1 & Sí\, x = x' \\ 0 & En\, otro\, caso \end{cases}$$

Razonamiento fuzzy

A diferencia de los sistemas expertos convencionales, donde las proposiciones (reglas) son ejecutadas en serie, el protocolo principal de razonamiento detrás de la lógica fuzzy es un paradigma de procesamiento paralelo. En los sistemas convencionales basados en conocimientos se aplican algoritmos de poda y /o heurísticos para reducir el número de reglas a examinar, pero en los sistemas fuzzy todas las reglas son disparadas.

El mecanismo raíz en un modelo fuzzy son las proposiciones. Estas son reglas que establecen relaciones entre las variables del modelo y una o más regiones fuzzy. Una serie de preposiciones fuzzy condicionales e incondicionales son evaluadas a través de su grado de verdad (pertenencia) y todas aquella que tengan algún grado de verdad contribuyen al estado final de la salida, del conjunto variable solución. El lazo funcional entre el grado de verdad y las regiones fuzzy relacionadas, es llamado el “método de implicación” . el lazo funcional entre las regiones fuzzy y el valor esperado de un conjunto objetivo, es denominado el “método de defuzificaciòn” . tomándolos juntos, ellos se constituyen en el sostén (columna vertebral) del razonamiento aproximado.

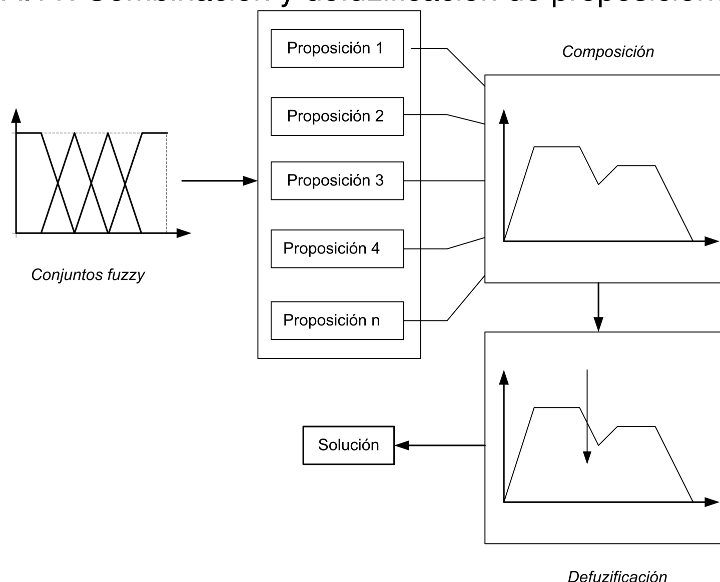
Mientras se están desarrollando soluciones a un método fuzzy, el sistema de razonamiento fuzzy convierte cada variable solución dentro de una región fuzzy provisional, así:

$$z_i \Rightarrow Z_i$$

donde z es la variable solución (algunas veces llamada la salida) y Z es el correspondiente conjunto fuzzy solución. A medida que cada proposición es evaluada, su consecuente región fuzzy es usada para actualizar la región fuzzy solución. Veasé figura A.11.

Este proceso de actualización esta bajo el control de una función de transferencia (g) que implementa una regla de implicación entre el estado de la región fuzzy consecuente y el estado de la salida fuzzy así:

Figura A.11: Combinación y defuzificación de proposiciones fuzzy



Fuente: Autores

$$g(w_i) \Rightarrow Z_i$$

Existen muchas posibles funciones de transferencias para la implicación, pero en ultimas cada una intenta correlacionar el sistema semántico del antecedente con la semántica del consecuente, y así generar una solución compatible con el significado del estado fuzzy para cada variable de salida.

Proposiciones fuzzy

Un modelo fuzzy consiste de una serie de condicionales e incondicionales proposiciones fuzzy. Una proposición establece una relación entre un valor en el dominio fundamental y un espacio fuzzy. Las proposiciones son expresadas en la forma,

$$x\, es\, Y$$

donde x es un escalar del dominio y Y es una variable lingüística. El resultado de la evaluación de una proposición fuzzy es un rango o grado de pertenencia (membresía) derivado de la función de pertenencia,

$$\mu_a \Leftarrow (X \in Y)$$

El grado de pertenencia derivado establece una compatibilidad entre x y el espacio fuzzy Y. tales proposiciones responden la pregunta ¿Qué tan compatible es x con Y ? ¿Con qué grado es x un miembro del conjunto Y?. Este valor se usa en las funciones de transferencias de correlación e implicación para crear o actualizar el

espacio fuzzy solución de salida. El espacio fuzzy solución final, se crea agregando las proposiciones fuzzy coleccionadas o correlacionadas. La correlación se basa en el valor de verdad de cada proposición fuzzy.

Proposiciones fuzzy condicionales

Una proposición condicional es aquella que se encuentra cualificada por una palabra *si* (*if*). Son análogas a las reglas de un sistema experto convencional. La proposición tiene la forma general

Si w es Z entonces x es Y

Donde w y x son valores escalares del modelo, Z y Y son variables lingüísticas. La proposición que sigue al termino es *si* es el antecedente o predicado y es alguna proposición fuzzy arbitraria. La proposición que sigue al termino *entonces* es el consecuente y es también alguna proposición fuzzy arbitraria. La proposición x es Y esta condicionada al valor de verdad del predicado. Se podría interpretar lo anterior de la siguiente forma:

x es miembro de Y con el grado en que w es un miembro de Z.

Esto es, el consecuente esta correlacionado con el valor de verdad del antecedente. La proposición fundamental puede ser extendida usando conectores fuzzy

Si (w es Z)•(y es W)•.......(u es S) entonces x es Y

Donde el (•) es alguno de los operadores *and* u *or*.

Proposiciones fuzzy incondicionales

Una proposición fuzzy incondicional es aquella que no se encuentra cualificada por una palabra SI (IF). La proposición tiene la forma general

$$x \, es \, Y$$

donde x es un escalar del dominio y Y es una variable lingüística. Las proposiciones incondicionales son aplicadas con frecuencia dentro de los modelos y dependiendo de cómo ellas sean aplicadas, sirven para restringir el espacio de salida o para definir un espacio solución por defecto, para el caso en que ninguna de las reglas condicionales se ejecute.

Orden de ejecución de las proposiciones

Para aquellos modelos que contiene solamente proposiciones condicionales o solamente proposiciones incondicionales, el orden en el cual las proposiciones (reglas) son ejecutadas no es importante. Sin embargo, si un modelo contiene una mezcla de estos dos tipos, entonces el orden de ejecución llega a ser importante. El efecto de aplicar proposiciones incondicionales cambia la naturaleza del espacio solución del modelo dependiendo de si las proposiciones son aplicadas antes o después del conjunto de proposiciones condicionales.

Las proposiciones incondicionales son usadas generalmente para establecer el conjunto soporte por defecto del modelo. Si ninguna de las reglas condicionales se ejecuta, entonces un valor para la variable solución es determinado del espacio delimitado por las reglas incondicionales. Por esta razón ellas tienen que ejecutarse antes que las condicionales. Si ninguna de las reglas condicionales que caen dentro del mismo dominio fundamental de las incondicionales, tiene un antecedente de intensidad más grande que el máximo de la intersección de las reglas incondicionales, ellas no contribuirán a la solución del modelo.

Aunque, comunmente menos utilizada, las reglas incondicionales pueden ser usadas para restringir el espacio solución final al modelo, al máximo valor de verdad de sus intersecciones. Esto se hace aplicando las reglas incondicionales después de que todas las propoposiciones condicionales han sido evaluadas.

Razonamiento proporcional

Comenzaremos la exploración del razonamiento fuzzy con el simple método monótonico, una técnica básica de implicación fuzzy.Aunque esta forma de razonamiento no es usada a menudo como método de implicación primario, esta es usada con frecuencia en el método de encadenamiento métrico del escalamiento fuzzy. Cuando dos regiones fuzzy están relacionadas a través de una simple función de implicación proporcional

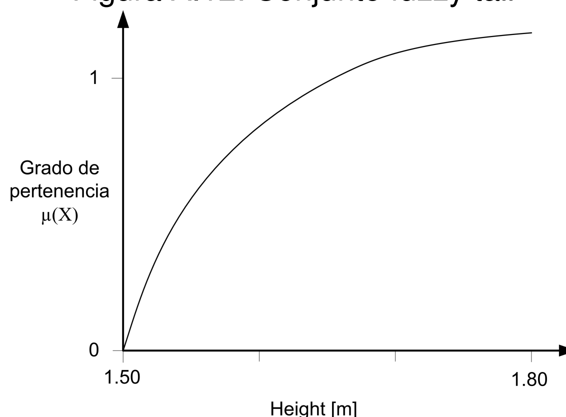
$$Si\, x\, es\, Y\; entonces\, z\, es\, W$$

Funcionalmente representada por la función de transferencia

$$Z = f((x, Y), W)$$

Entonces bajo un conjunto limitado de circunstancias, un sistema de razonamiento fuzzy puede desarrollar un valor esperado sin pasar a través de los procesos de composición y defuzificación . El valor de la salida es estimado directamente del correspondiente valor de verdad de las regiones fuzzy del antecedente.

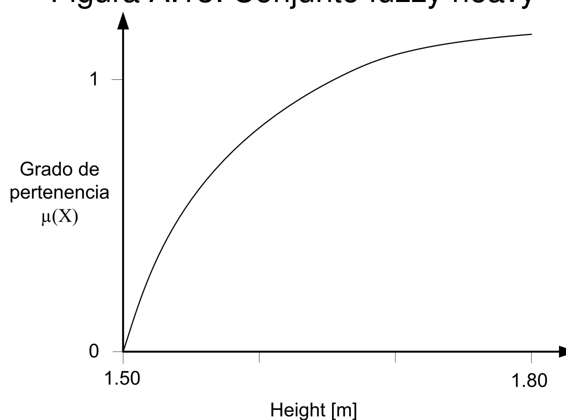
Figura A.12: Conjunto fuzzy tall



Fuente: Autores

Esta forma de inferencia fuzzy obedece a un método de implicación llamado “selección monótona”. Para ilustrar como trabaja, considerese un simple modelo de estimación del peso. El vocabulario fuzzy fundamental para este modelo consiste de dos conjuntos fuzzy *alto* (Figura A.12) y *pesado* (Figura A.13).

Figura A.13: Conjunto fuzzy heavy



Fuente: Autores

Estos dos conceptos fuzzy proveen la base para el modelo de estimación del peso. El modelo esta basado en la percepción generalizada de quqe existe una relación entre la estatura de los individuos y su peso, donde el peso se expresa como una única regla

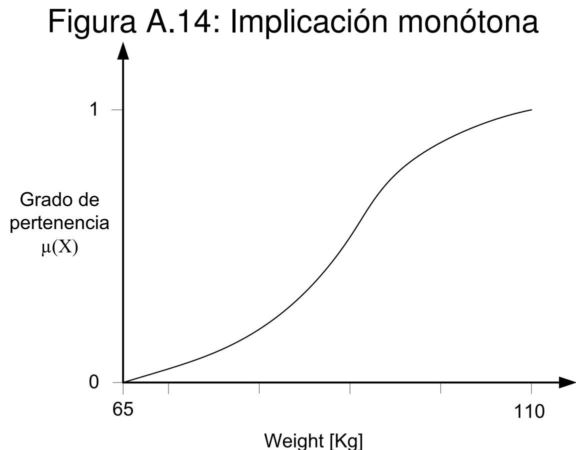
Si estura es tall entonces peso es heavy

Una implicación de selección monótona entre las regiones fuzzy Y y W se basa en el siguiente algoritmo de selección:

- Para un elemento x en el algoritmo Y, encontrar su pertenencia en la región fuzzy Y. Esto es $\mu_Y[x]$.

- En al región fuzzy W con el valor de pertenencia correspondiante a $\mu_Y[x]$, interceptamos la superficie fuzzy. Bajamos, desde el punto de intersección , una linea recta hacia el dominio. El valor en el eje Y dominio, a es el valor de la función de implicación. Funcionalmente esto es expresado como:$Z_w =$f($\mu_y[x]$, Dw)

Figura A.14: Implicación monótona



Fuente: Autores

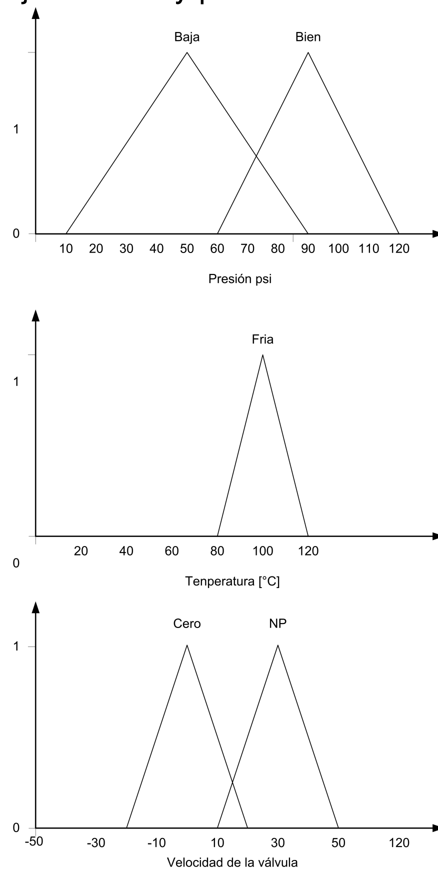
La figura A.14, ilustra el proceso de implicación para este modelo cuando la estructura es de 1.5 m, su pertenencia en el conjunto fuzzy *alto* es de [0.36]. Este valor es usado con el conjnto fuzzy *pesado* para la varible solución. Se "entra" al conjunto fuzzy *pesado* con el valor de [0.36] en el eje de valores de pertenencia y nos desplazamos horizontalmente hasta encontrar la superficie del conjunto. Un valor para la variable solución peso, es encontrado en el dominio del conjunto *pesado*, para este valor de pertenencia.

En conclusión el razonamiento proporcional (monótono) nos proporciona un poderoso método para encadenar los valores de pertenencia de dos regiones fuzzy generales. Esto nos permite hacer estimaciones acerca de la estructura del dominio de la región fuzzy X cuando conocemos el dominio y el valor de pertenencia de un punto en la región fuzzy Y. Sin embargo el razonamiento monótono es menos general y más restrictivo que el método convencional de implicación de reglas *min/max*. Lo anterior es el resualtado de dos limitaciones: primero se requiere que la salida para el modelo sea una variable fuzzy simple controlada por una simple regla fuzzy, y segundo la función de implicación entre dos regiones fuzzy es expresado como una correlación entre las topologías de la superficie. Tanto como se incrementa la complejidad del predicado de la proposición (regla), el grado con el cual el razonamiento monótono podría producir resualtados consistentemente validos, tiende a disminuir. Esto es una consecuencia de las bases de la teoría de Complejidad e información.

Reglas composicionales fuzzy de inferencia

A diferencia del razonamiemto monótono, el espacio de implicación generado por las reglas composicionales de inferencia, se deriva de la agregación y correlación de espacios fuzzy producidos por la interacción de muchas reglas (proposiciones). En efecto, todas las proposiciones están corriendo en paralelo para crear un espacio de salida que contiene información de todas las proposiciones. Existen dos métodos principales de inferencia en los sistemas fuzzy. Estos métodos difieren en la manera en que actualiza la representación del espacio fuzzy de salida.

Figura A.15: Conjuntos fuzzy para variables de entrada y salida



Fuente: Autores

Reglas *MIN-MAX* de implicación

La operación composicional *Min-Max* deriva su nombre del método que aplica. La región fuzzy consecuente se restringe al mínimo valor de verdad del predicado. La región fuzzy de salida es actualizada tomando el máximo de esos conjuntos fuzzy minimizados.

Reglas aditivas fuzzy

La operación composicional aditiva tiene una ligera diferencia en la forma de actualizar la región fuzzy de la derivada solución (de salida). La región fuzzy consecuente se mantiene reducida por el mínimo valor de verdad del predicado, pero la región fuzzy de salida es actualizada por una regla diferente. Esta es esencialmente una operación de adición limitada aplicada a la región fuzzy de salida. En lugar de tomar el máx $(\mu_A[x_i], \mu_B[x_i])$ en cada punto a lo largo del conjunto fuzzy de salida, se suman los valores de funciones de pertenencia. La adición está delimitada por [1.0] de forma que el resultado de cualquier adición no pueda exceder el máximo valor de verdad de un conjunto fuzzy.

Ejemplo de implicación fuzzy con reglas composicionales

Para efectos del presente ejemplo se usa un modelo con tres reglas (proposiciones) y dos variables fuzzy de entrada: Presión y temperatura y una variable fuzzy de salida: velocidad-válvula.

Para la variable Presión se manejarán dos términos lingüísticos: *bajo* y *bien*. Para la variable temperatura el termino lingüístico *fria*. Para la variable velocidad-válvula los términos lingüísticos *cero* y *positivamente-moderado (pm)*. En la figura A.15, se pueden observar los conjuntos fuzzy para los anteriores conceptos. Se busca la región fuzzy de salida, cuando la presión sea de 70 psi y la temperatura 120ºC las tres reglas que se aplicaran son:

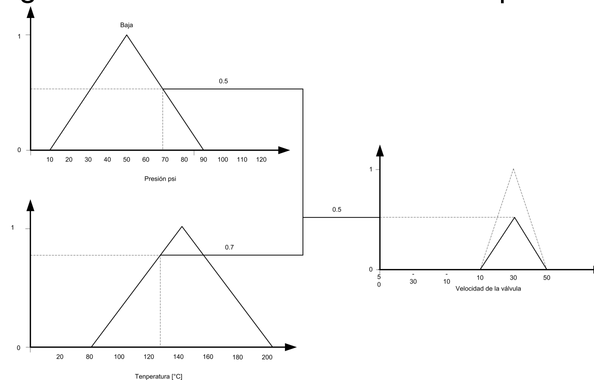
1. Sí la presión es *baja* y la temperatura es *fria* entonces velocidad-válvula es *pm*.
2. Sí la presión es *bien* entonces velocidad-válvula es *pm*.
3. Sí la presión es *bien* y la temperatura es *fria* entonces la velocidad-válvula es *cero*.

Métodos de correlación

Los mecanismos de implicación *Min/Max* y adición fuzzy utilizados para las proposiciones condicionales fuzzy, involucran una reducción del valor de verdad de la región fuzzy consecuente basada en el valor de verdad de las reglas, antes que la región fuzzy de la variable solución que se esta trabajando, sea actualizada. Existen dos métodos principales para restringir la altura del conjunto fuzzy consecuente: Correlación mínima y Correlación producto.

- Correlación mínima: Es el método más común de correlacionar el consecuente con el valor de verdad de la premisa, truncando (cortado) la región fuzzy a la altura del valor verdad de la premisa. El método se denomina de correlación mínina, puesto que el conjunto fuzzy es minimizado, trucándolo a la altura del valor de verdad mínimo del predicado. Este método de correlaciones fue el que se utilizó anteriormente para desarrollar el ejemplo.

Figura A.16: Método de correlación producto



Fuente: Autores

Usualmente el mecanismo de correlación mínima crea una meseta, dado que la cima de la región fuzzy es cortada a la altura del valor de verdad del predicado. Lo anterior introduce una cierta cantidad de pérdida de la información. Si el conjunto fuzzy es multimodal, o bien irregular, la topología de la superficie por encima del nivel del valor de verdad, el predicado es descartado. El método de correlación mínima, sin embargo es a menudo preferido sobre el de correlación producto (el cual preserva la forma de la región fuzzy) puesto que reduce intuitivamente el valor de verdad del consecuente al nivel del mínimo valor del predicado, involucrando menos complejidad y procesamiento aritmético más rápido (una consideración muy importante a nivel de microprocesadores y microcontroladores), y casi siempre genera una superficie agregada de salida que es más fácil de defuzificar usando las técnicas convencionales de centro de momentos (centroide) o centro de máximos.

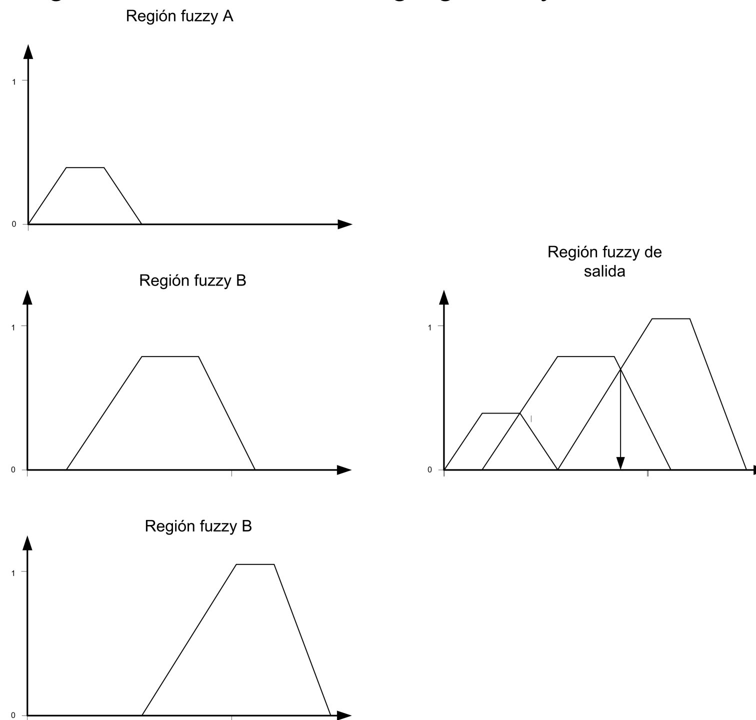
- Correlación producto: Con este método la región fuzzy intermedia (consecuente) es escalada en vez de truncada, o sea, la función de pertenencia se escala a la altura del valor predicado. Lo anterior tiene el efecto de escoger la región fuzzy mientras retiene aun la forma original del conjunto fuzzy. En la figura A.16, se muestra como opera el método aplicando a la regla (Min-Max).

Métodos de descomposición y defuzificación

Usando las reglas generales de una inferencia fuzzy, la evaluación de una proposición produce un conjunto fuzzy con cada variable solución del modelo. Por ejemplo, las siguientes proposiciones, cuando sean evaluadas, correlacionaran los consecuentes conjuntos fuzzy A, B y C para producir el conjunto fuzzy representativo de la variable solución D.

Si w es Y entonces D es A
Si x es X entonces D es B

Figura A.17: Proceso de agregación y defuzificación



Si y es Z entonces D es C

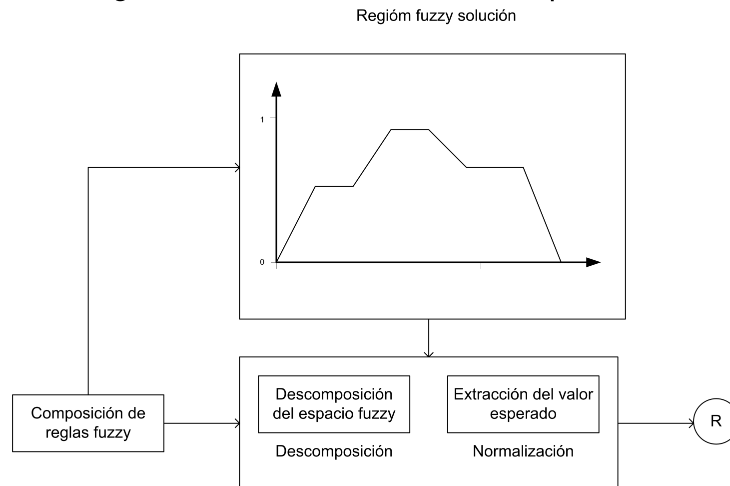
Para encontrar el valor actual para la correspondiente escalar *d*, se tendrá que encontrar un valor que represente mejor la información contenida en el conjunto fuzzy *D*. Este proceso, ilustrado en la figura A.17, se denomina defuzificacion y produce el valor esperado de la variable de salida para una ejecución (estado) particular del modelo fuzzy.

La defuzificacion es la fase final del razonamiento fuzzy. Como se ilustra en la figura A.18, la evaluación de las proposiciones del modelo es manejada a través de un proceso de agregación que produce las regiones fuzzy finales para cada variable solución. Estas regiones son entonces descompuestas usando uno de los métodos de defuzificacion.

Para los modelos fuzzy, existen varios métodos para determinar el valor esperado de la región fuzzy solución. Estos son los “métodos de descomposición”, denominados también “métodos de defuzificacion”, y describen las formas en que se puede extraer (encontrar) un valor esperado para un estado del espacio fuzzy final. El entendimiento actual de las reglas de descomposición, son de origen mas heurístico que algorítmico, y proviene de los “primeros principios”. Esta es probablemente la mejor manera de poder determinar un valor de la región fuzzy solución, puesto que no parece posible poder representar un espacio multidimensional y complejo, mediante un simple número.

Como se muestra en la figura A.19, la defuzificación permite dejar caer una linea

Figura A.18: Proceso de descomposición



Fuente: Autores

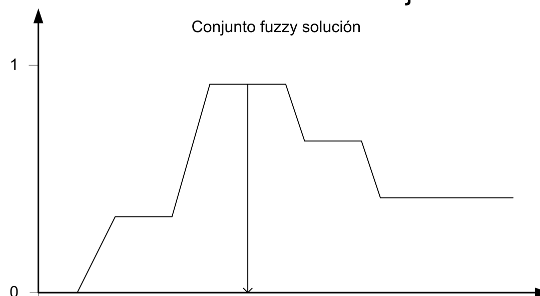
recta hacia el dominio. En el punto donde esta linea corta el eje del dominio, se lee el valor esperado del conjunto fuzzy solución, para la variable de salida.

Centro de momento (centroide)

La técnica del centroide, del centro de momento o del centro de gravedad, encuentra el punto de balance de la región fuzzy solución, calculando el peso medio de la región fuzzy. Aritméticamente, para la región fuzzy solución *A*, esta formulado como.

$$R = \frac{\sum_{i=0}^{n} d_i \mu_A(d_i)}{\sum_{i=0}^{n} \mu_A d_i}$$

Figura A.19: Defuzificación de un conjunto fuzzy solución



Fuente: Autores

Donde *d* es el i-esimo valor del dominio y $\mu(d)$ es el valor de pertenencia para ese punto del dominio. Se puede decir entonces que la defuzificacion por el centro de momento o centroide , encuentra un punto que representa el cetro de gravedad del conjunto fuzzy.

La técnica del centro de gravedad es la más usada dado que tiene unas propiedades muy importantes:

- Los valores defuzificados tienden a moverse suavemente alrededor de la región fuzzy de salida, esto es, cambios en la topología de un conjunto fuzzy de un modelo construido al siguiente, usualmente generan pequeños cambios en el valor esperado.
- Este método es relativamente fácil de calcular.
- Puede aplicarse a conjuntos fuzzy tanto geométricos como de tipo singleton o discretos.

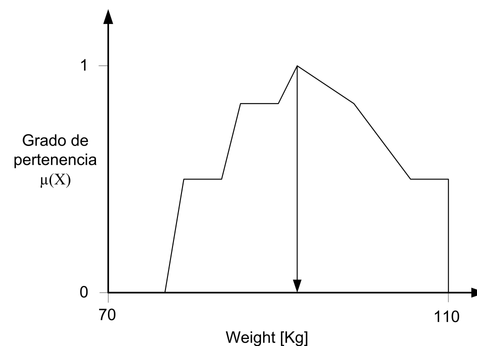
Altura máxima

Existen tres categorías, estrechamente relacionadas con la técnica de la altura máxima: El promedio de máximos, el centro de máximos y la altura máxima simple. A diferencia de la técnica del centroide, la descomposición por altura máxima tiene algunos atributos que son generalmente aplicables a una muy reducida clase de problemas, estos son:

1. El valor esperado es sensitivo a una simple regla que domine el conjunto de reglas fuzzy .
2. El valor esperado tiende a saltar de un modelo construido al siguiente tanto como la forma de la región cambie.

- Altura máxima simple:Este método (ver figura A.20), encuentra el punto del dominio que tiene el máximo valor de verdad. Si este punto es ambiguo (esto es, el máximo es una meseta) entonces se usa otro método asociado para resolver el conflicto: promedio de máximos.

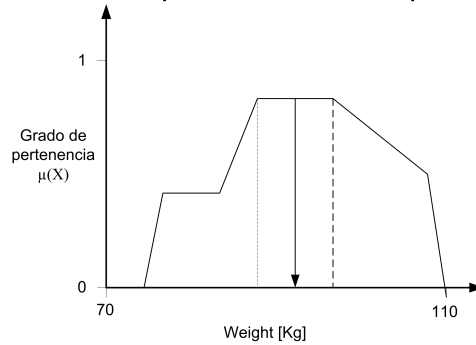
Figura A.20: Defuzificación con el método de altura máxima simple



Fuente: Autores

- Promedio de máximos: Este método (ver figura A.21), encuentra la media del valor máximo de la región fuzzy. Si el máximo es un simple punto, retorna dicho valor; de otra manera el promedio de la meseta es calculado y retornado.

Figura A.21: Defuzificación por el método de promedio de máximos

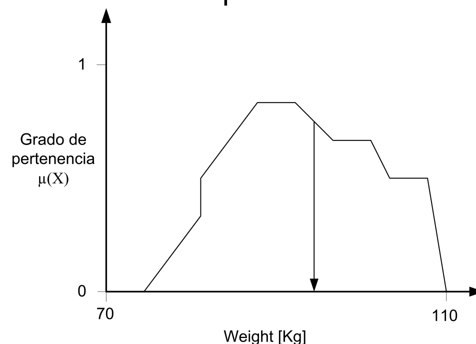


Fuente: Autores

- Centro de máximos: En una región fuzzy multimeseta, este método encuentra la meseta más alta y la siguiente meseta más alta. El punto medio entre los centros de estas dos mesetas es seleccionado. Veasé figura A.22.

Existen otros métodos para descomponer o defuzificar un conjunto fuzzy hacia un valor esperado. Sin embargo, los métodos presentados anteriormente son las formas de defuzificación usadas más ampliamente. En particular, el método de centro de momentos es el preferido en la gran mayoría de los modelos, puesto que este parece asimilar toda la información contenida en el conjunto fuzzy de salida. A menos que se tenga razones para creer que el modelo requiere de métodos de defuzificacion más avanzados o especializados.

Figura A.22: Defuzificación por método de centro máximos

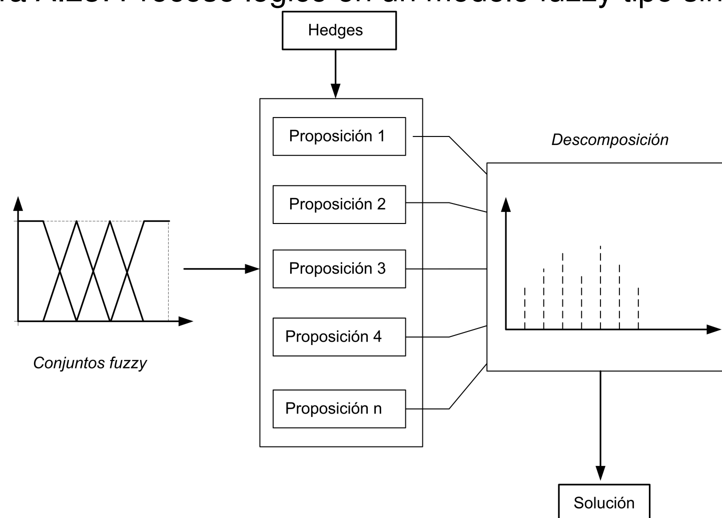


Fuente: Autores

Representación geométrica tipo singleton

En el espacio de salida de una geometría singleton, los términos asociados con las regiones fuzzy son representados como simples puntos verticales, en vez de las típicas funciones de pertenencia de los conjuntos fuzzy. En secuencia se designa un único y abrupto valor soporte para la salida. El método de inferencia composicional para un modelo de geometría singleton es ligeramente diferente que el usado en aquellos basados en regiones fuzzy. Como se muestra en la figura A.23 , el paso denominado agregación es removido puesto que los singletones son proporcionalmente modificados en lugar de ser combinados.

Figura A.23: Proceso lógico en un modelo fuzzy tipo singleton



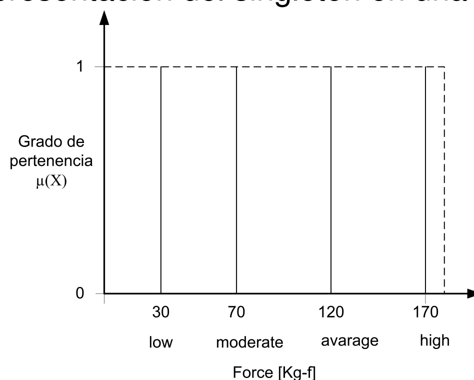
Fuente: Autores

El método de implicación usado en los singletones es generalmente el mismo que el usado con los conjuntos fuzzy, excepto que la función de transferencia es directa. Cada singleton representa un punto en el espacio de salida, y se conecta con sus vecinos para propósitos de interpolación, mediante interpolación lineal.

Los singletones son especificados como puntos soporte a lo largo del espacio de salida de la variable solución. Cada singleton, al igual que los términos lingüísticos de un conjunto fuzzy, es identificado por una etiqueta. La figura A.24 muestra como la variable de salida *fuerza* puede estar compuesta de dos puntos individuales únicos.

En la geometría singleton se hace la distinción entre el esquema representacional para el vocabulario soporte del modelo y el esquema solución de salida. Aún se necesita manejar las proposiciones como expresiones basadas en conjuntos fuzzy, en orden de producir el valor de verdad para el predicado. Sin embargo, una vez se tiene el valor de verdad para el predicado, este puede aplicarse directamente a la salida singleton como un factor de escalamiento. En la geometría singleton las

Figura A.24: Representación del singleton en una variable de salida

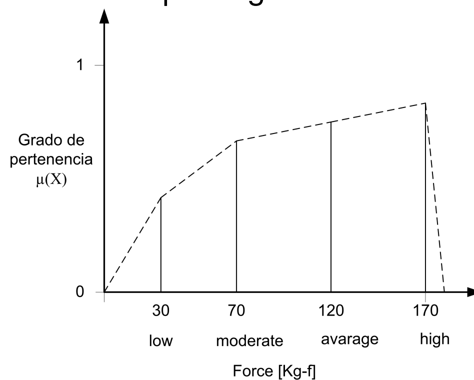


Fuente: Autores

proposiciones o reglas son expresadas en manera idéntica al modelo basado en conjunto fuzzy.

$$\text{Si } y_i \text{es } V_i \bullet x_i \text{ es } Y_i \text{ entonces } Z_i \text{ es } S_i$$

Figura A.25: Variable de salida tipo singleton escalada por valores de verdad



Fuente: Autores

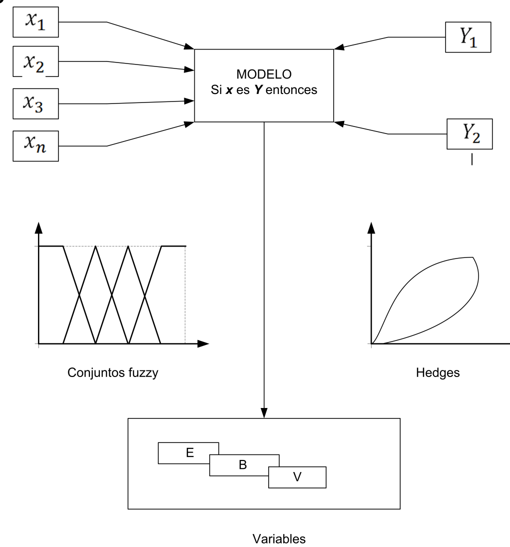
Donde S_i es algún termino lingüístico del tipo singleton. Cuando el modelo es lineanizado, cada punto singleton tiene un valor de [1.0] y su valor futuro es determinado por un escalonamiento proporcional debido al valor de verdad del predicado. La figura A.25 muestra como la variable de salida *fuerza* podría ser ajustada en esta forma. Puesto que una línea es trazada entre los puntos singleton, para construir una representación de conjuntos fuzzy de salida, la defuzificacion en singleton, y sistemas basados en conjuntos fuzzy es equivalente. De hecho, el método del centroide o centro de gravedad es ligeramente simplificado. El centroide se calcula como sigue:

$$R = \frac{\sum_{i=0}^{n} d_i \mu_A S_i}{\sum_{i=0}^{n} \mu_A S_i}$$

En este caso no se necesita realizar una integración numérica de toda la superficie del conjunto fuzzy. En lugar de ello, el valor del dominio de cada singleton es multiplicado por su altura. Esta técnica general de defuzificacion produce un centroide que es generalmente equivalente al centroide encontrado tomando el área a través del conjunto fuzzy entero.

Modelo fuzzy

Figura A.26: Modelado de un sistema fuzzy



Fuente: Autores

Un modelo fuzzy, al igual que en los tradicionales sistemas expertos y de soporte de decisiones, está basado en el concepto de flujo para un sistema con entrada, proceso y salida. Un modelo fuzzy difiere, sin embargo, en dos propiedades importantes:

- Que fluye dentro y fuera del proceso.
- La actividad fundamental de transformación embebida en el proceso mismo. La figura A.26 es un esquema de ambiente de un modelamiento fuzzy.

El modelo acepta un conjunto de entrada $(x_1, x_2, ..., x_n)$ como su conocimiento acerca del mundo exterior. Esas entradas pueden ser tanto escalares aritméticos ordinarios, vectores y matrices como también regiones fuzzy. Las entradas fuzzy podrían ser regiones fuzzy de salida de modelos previos, conjuntos fuzzy que estén globalmente disponibles a través del modelo, números fuzzy. El modelo por si mismo ejecuta una serie de proposiciones o reglas fuzzy, usando estructuras de control almacenados en el código de la aplicación. El modelo ejecuta esas proposiciones en una manera tal que simula procesamiento paralelo. Esto hace que los valores de

las variables solución no estén disponibles para su uso hasta que todas las reglas hayan sido ejecutadas.

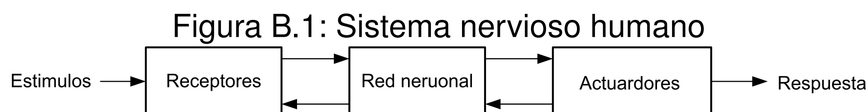
B. REDES NEURONALES

Una red neuronal artificial (RNA) es un modelo computacional inspirado en redes neuronales biológicas que puede ser consideradas como un sistema de procesamiento de información con características como aprendizaje a través de ejemplos adaptabilidad, robustez, capacidad de generalización y tolerancia a fallas. La RNA puede ser definida como una estructura distribuida, de procesamiento paralelo, formada de neuronas artificiales (llamadas también elementos de procesamiento o nodos), interconectados por un gran numero de conexiones (sinapsis), los cuales son usados para almacenar conocimiento, con el fin de que este disponible para poder ser usado.

El cerebro humano

El sistema nervioso humano es visto como un sistema de tres etapas, donde el centro del sistema es el cerebro, representado por una red neuronal, la cual continuamente recibe información, la percibe, y toma decisiones apropiadas. Los receptores convierten los estímulos del cuerpo humano o del ambiente en impulsos eléctricos que transportan información a la red neuronal. Los actuadores transportan los impulsos eléctricos generados por la red neuronal o cerebro, respondiendo como un sistema de salida como se observa en la figura B.1. La sinapsis son unidades funcionales y estructurales que promedian las interacciones entre las neuronas. El tipo más común de sinapsis es la química, la cual opera de la siguiente manera:

Figura B.1: Sistema nervioso humano

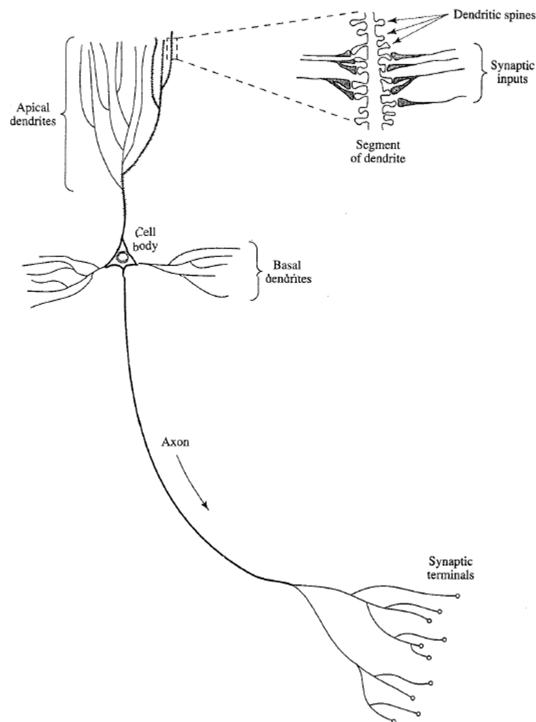


Fuente: Autores

Un proceso pre-sináptico libera una sustancia transmisora que se difunde a través de los empalmes sinápticos entre las neuronas y luego actúan en un proceso post-sináptico. Así de este modo, una sinapsis convierte una señal eléctrica en una química, y luego de nuevo en una señal eléctrica post-sináptica. La plasticidad permite que el sistema nervioso central se adapte a su alrededores. En un cerebro adulto, la plasticidad se puede dar por dos mecanismos. La creación de nuevas conexiones sinápticas entre neuronas y por la modificación de sinapsis existentes. Los Axones que son las lineas de transmisión, y la dendritas que son las zonas receptivas, son dos tipos de células que son distinguidas por su morfología, un axón tiene una superficie más lisa, menos ramificada y de mayor longitud que cualquier dendrita que tiene una superficie irregular y ramificada. Las neuronas vienen de una amplia variedad de formas y tamaños en diferentes partes del cerebro.

la figura B.2 muestra una figura de una célula en forma de pirámide. La mayoría de las neuronas codifican sus salidas como una serie de pulsos de voltaje breves, estos pulsos comúnmente conocidos como *potencial de acción* originados en el interior del cuerpo de las neuronas y luego se propagan a través de cada neurona individual a una velocidad y amplitud constante, la razón del uso de los potenciales

Figura B.2: Celula piramidal



de acción en la comunicación entre neuronas se basa en el aspecto físico del axón. El axón de una neuronal es largo y delgado lo cual es característico de alta resistencia y gran capacitancia, Este entonces podría ser una linea de transmisión RC. El análisis del mecanismo de propagación en el axón revela que cuando el voltaje es aplicado en un extremo de él, este cae exponencialmente con la distancia, en cantidades insignificantes hasta alcanzar el otro extremo de el axón.

Se estima que el cerebro humano tiene mas de cien mil millones de neuronas y 10^{14} sinapsis en el sistema nervioso. Los estudios realizados sobre la anatomía del cerebro humano concluyen, que hay en general más de mil sinapsis por término medio en cada entrada y salida de cada neurona. Aunque el tiempo de conmutación de las neuronas biológica es casi un millón de veces mayor que que en los actuales componentes de las computadoras, la conectividad de las neuronas naturales es de mil veces la de las neuronas artificiales. En conclusión el objetivo principal de las neuronas biológicas es desarrollar aplicaciones de síntesis y procesamiento de información.

Las redes neuronales artificiales son un procesador masivo distribuido en paralelo constituido por unidades simples de procesamiento (nodos), las cuales tienen una tendencia natural de almacenamiento de conocimiento experimental para tenerlo disponible para algún especifico. Estas se parecen al cerebro en dos aspectos:

- El conocimiento es adquirido por la red del ambiente a través de procesos de aprendizaje.
- La fuerzas de las interconexiones neuronales, conocidos como pesos sinápticos, son usados para el almacenamiento del conocimiento adquirido.

El procedimiento para lograr el aprendizaje de la red se llama algoritmo de aprendizaje, cuya función es modificar los pesos sinápticos de una manera ordenada para alcanzar el objetivo deseado del diseño.

Ventajas de las Redes Neuronales

Las redes neuronales proporcionan las siguientes utilidades y capacidades:

- No linealidad: Una neurona artificial puede ser lineal o no lineal. Una red neuronal esta constituida por interconexiones de neuronas no lineales. Además la no linealidad es de un tipo especial, porque esta es distribuida a través de la red. La no linealidad es una propiedad de alta importancia, sí la fuente física responsable de la generación de las señales de entrada es no lineal.
- Aprendizaje a partir de ejemplos: Un paradigma popular de aprendizaje llamado aprendizaje con un profesor o aprendizaje supervisado modifica los pesos sinápticos de una red neuronal por la aplicación de un conjunto de entrenamientos. Cada ejemplo consiste de una única señal de entrada y su correspondiente respuesta deseada. Se le presenta un ejemplo escogido al azar de un conjunto, a la red neuronal, y los pesos sinápticos de la red son modificados para minimizar la diferencia entre la respuesta deseada y la respuesta actual de la red. El entrenamiento de la red es repetido para muchos ejemplos hasta que esta alcanza un estado seguro donde no haya un cambio significativo en los pesos sinápticos.
- Adaptabilidad: Una red neuronal artificial tiene la capacidad de adaptar sus pesos sinápticos cuando cambia su ambiente, en particular una red entrenada para operar en un ambiente especifico, puede ser fácilmente entrenada de nuevo para operar con pequeños cambios en las condiciones de operación. Además cuando se require que la RNA opere en unas condiciones no estacionarias, la red puede ser diseñada para cambiar sus pesos sinápticos en tiempo real. La arquitectura natural de una RNA para clasificación de patrones de datos, procesamiento de señales, y control de procesos esta relacionada con la capacidad de adaptabilidad de la red. Como regla general, se puede decir que sí el sistema tiene mayor adaptabilidad, su estabilidad sera mejor.

- Evidencia de la respuesta: En el contexto de los patrones de clasificación, una red neuronal puede ser diseñada no solamente para proporcionar información acerca del patrón seleccionado, si no también de la con-fiabilidad de la decisión tomada. Esta información puede ser usada para rechazar patrones ambiguos que pueden surgir, y de este modo mejorar el rendimiento de clasificación de la red.

- Información contextual: El conocimiento es representado por varias estructuras y activaciones de estado de la red. Toda neurona de la red es potencialmente afectada por la activación global de las demás neuronas de la red. Consecuentemente, la información contextual es manejada de una forma natural por una RNA.

- Tolerancia a fallas: Una RNA implementada en hardware tiene el potencial de tolerar fallas, o capaz de manejar arquitectura de computación robusta en el sentido que su rendimiento se degrada poco bajo condiciones adversas de operación, por ejemplo si la conexión de una neurona es dañada, la memoria es afectada en calidad, sin embargo debido a la distribución natural de la información almacenada en la red, el daño debe ser amplio para poder afectarla seriamente, mas generalmente la RNA muestra una disminución de su rendimiento en vez de una falla catastrófica.

- Implementable: El natural procesamiento en paralelo que tiene la RNA hace que sea potencialmente rápido a la hora de computar ciertas tareas.

- Uniformidad de análisis y diseño: Básicamente las redes neuronales disfrutan de universalidad como procesador de información, en el sentido que la misma notación es usada en todos los dominios de aplicación de las redes neuronales. Estas características se manifiestan de diferentes maneras:

  - Neuronas: Representan de una u otra forma un ingrediente común a todas las redes neuronales.
  - Estas cosas en común hace posible que se comparta teorías y algoritmos de aprendizaje en diferentes aplicaciones de las redes neuronales.
  - Las redes modulares pueden construirse a partir de la integración de módulos.

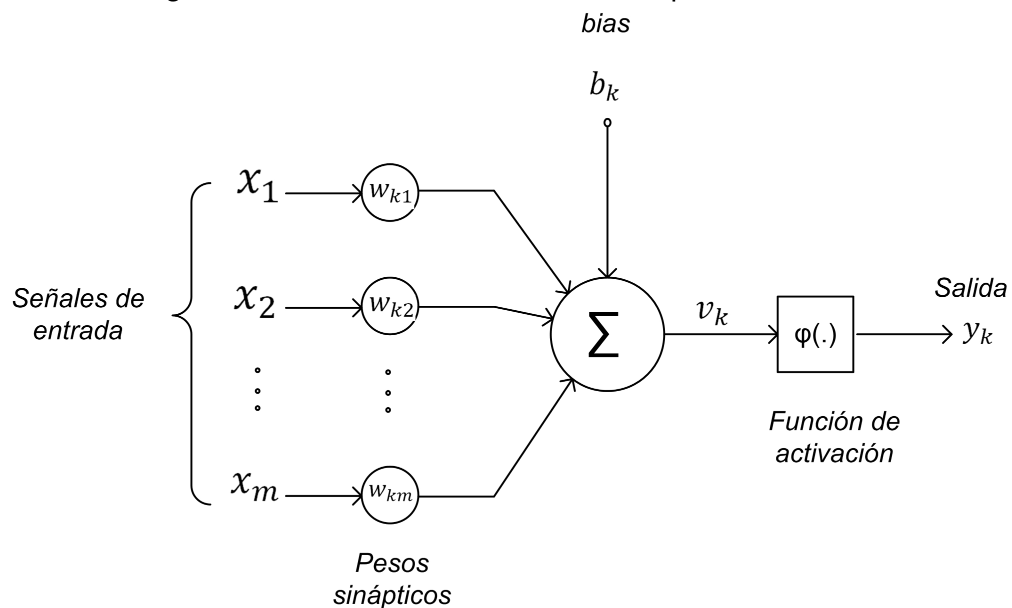
- Analogía con el cerebro humano: El diseño de una red neuronal esta inspirado en el cerebro humano, el cual es una prueba viviente de la tolerancia a fallas del procesamiento en paralelo no solamente físicamente, si no también en rapidez y potencia.

Modelo de una neurona

Una neurona es una unidad de procesamiento de información que es un fundamental en la operación de las redes neuronales. El diagrama de bloques de la figura B.3 muestra el modelo de una neurona, la cual esta conformado por tres elementos básicos:

- Un conjunto de sinapsis o conexiones, donde cada una se caracteriza por un peso o fuerza de la conexión, específicamente una señal x_j a la entrada de la sinapsis *j* conectada a una neurona *k* es multiplicada por el peso sináptico w_{kj}, donde el primer subíndice se refiere a la neurona en cuestión y el segundo hace referencia a la entrada. A diferencia de las sinapsis en el cerebro humano, los pesos sinápticos de una neurona artificial puede tomar tanto valores positivos como negativos.

Figura B.3: Modelo de una unidad de procesamiento



Fuente: Autores

- Un sumador que suma las señales de entrada por los pesos de la respectiva sinapsis de cada neurona.

- Una función de activación para limitar la amplitud de salida de una neurona. Esta función de activación se refiere a una *squashing function* que limita el rango de amplitudes permisibles de la señal de salida a un valor finito. Típicamente el rango de amplitudes normalizados a la salida de una neurona es el intervalo cerrado [0,1] o alternativamente [-1,1].

El modelo de la figura también incluye la aplicación externa de *bias*, denotada como b_k. Las bias tienen un efecto de incremento o reducción de la entrada neta a la función de activación , dependiendo de sí esta es positiva o negativa.En términos matemáticos se podría escribir para una neurona *k* lo siguiente: $u_k = \sum_{j=1}^{m} w_{kj} x_j$ y $y_k = \varphi(u_k + b_k)$ donde $x_1, x_2, ..., x_m$ son señales de entrada, $w_{k1}, w_{k2}, ..., w_{km}$ son pesos sinápticos de la neurona *k*, u_k es la combinación lineal de las señales de entrada, b_k son las bias, $\varphi(.)$ es la función de activación, y y_k es la señal de salida *de* la neurona. El uso de las bias b_k tiene el efecto de afinación de la salida u_k en el modelo.

$$v_k = u_k + b_k$$

Tipos de funciones de activación

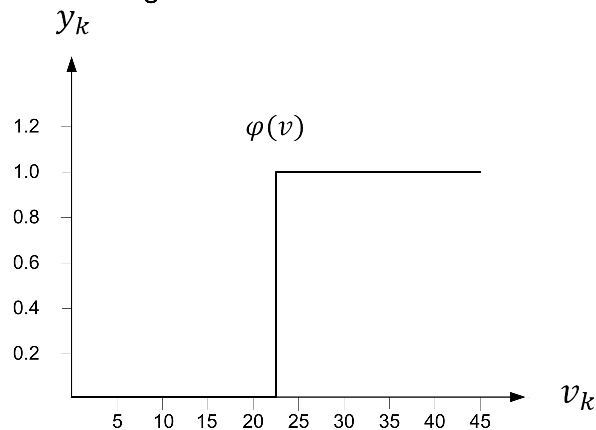
La función de activación denotada por $\varphi(v)$, define la salida de una neurona en terminos del campo local inducido, hay tres tipos básicos de función de activación.

- Función umbral. Para es te tipo de función de activación, mostrada en la figura B.4 tenemos:

$$\varphi(v) = \begin{cases} 1 & \text{if } v > 0, \\ 1 & \text{if } v = 0, \\ 0 & \text{if } v < 0. \end{cases}$$

  Donde cada neurona es referida a la literatura del *modelo Pitts McCulloch,* en reconocimiento a sus trabajos hechos en 1943. En este la salida de la neurona toma un valor de 1, sí el campo local de la neurona no es negativo, en otro caso toma 0. También se conoce como el modelo del todo o nada.

Figura B.4: Función umbral



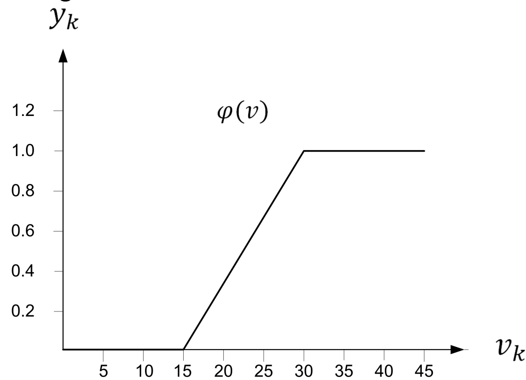
Fuente: Autores

- Función lineal a trozos: Para esta función descrita en la figura B.5, tenemos que:

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0{,}5 \\ v & \text{if } 0{,}5 > v > -0{,}5 \\ 0 & \text{if } v \leq -0{,}5 \end{cases}$$

donde el factor de ampliación influye en la región lineal, donde se supone que va

Figura B.5: Función lineal a trazos



Fuente: Autores

a estar la unidad. Esta función de activación puede ser vista como una aproximación a una amplificador lineal. Las siguientes situaciones pueden ser vistas como dos formas de función de activación a trozos.

- Función sigmoidal: Es la función de activación mas usada en la creación de redes neuronales artificiales. Es definida como una función de incremento estricto que muestra un buen balance entre comportamiento lineal y no lineal. La figura B.6 es un ejemplo de una función sigmoidal, es la función logística que se define como:

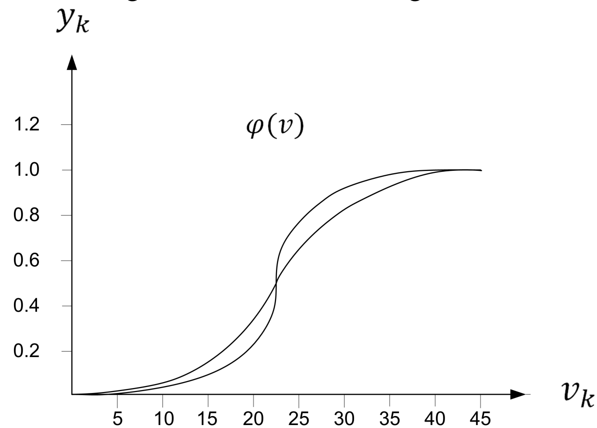
$$P = \frac{1}{1 + exp(-av)}$$

donde *a* es la pendiente de la función sigmoidal, si variamos *a* obtenemos funciones sigmoidales de diferente pendiente. Con valores de la pendiente limites la función sigmoidal empieza a simplificarse a una función umbral, pero mientras una función umbral da valores de 0 o 1, la función sigmoidal da un rango continuo de 0 a 1, esto quiere decir que es diferenciable.

Arquitectura de las redes neuronales

La manera con que las neuronas se organizan en la estructura de una red, esta íntimamente ligado al algoritmo de aprendizaje en el entrenamiento de la red, En

Figura B.6: Función sigmoidal



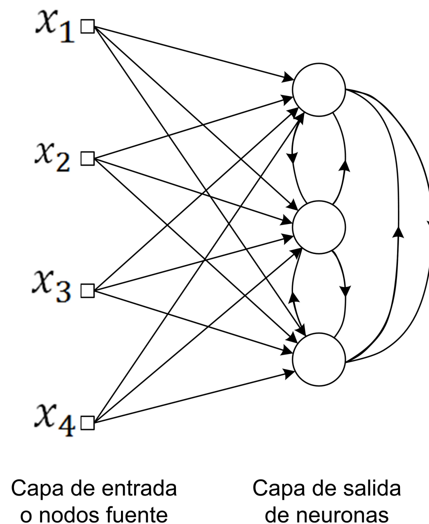
Fuente: Autores

general se puede identificar tres tipos de arquitectura de RNA.

Redes de una capa con feedforward.

En una red neuronal, las neuronas están organizadas en forma de capas. En la forma más simple de una red se tiene una capa de entrada de nodos fuentes que que se proyectan a las neuronas de la capa de salida, pero no viceversa, en otras palabras esta red es acíclica. Se dice que esta red es de una sola capa por su capa de salida, ya que la capa de entrada no se cuenta porque no hace ningún proceso.

Figura B.7: Red Neuronal de una capa con feedforward

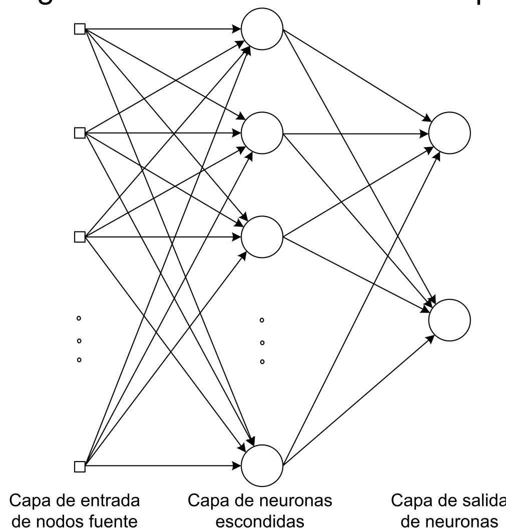


Fuente: Autores

Redes Neuronales multicapa

El segundo tipo de redes neuronal alimentadas hacia adelante se reconoce por la presencia de una o más capas, llamadas *capas escondidas*, y cuyas unidades de procesamiento son llamadas *neuronas escondidas* o *unidades escondidas*. La función de estas neuronas es intervenir entre las entradas externas y las salidas de la red de una manera indicada, por medio de la adición de una o más capas a la red, estas quedan queda a disposición de extraer datos estadísticos de orden más altos, esta habilidad de las unidades escondidas es muy importante cuando la cantidad de entradas a la red son grandes. Los nodos fuente de la capa de entrada suministran

Figura B.8: Red neuronal multicapa



Fuente: Autores

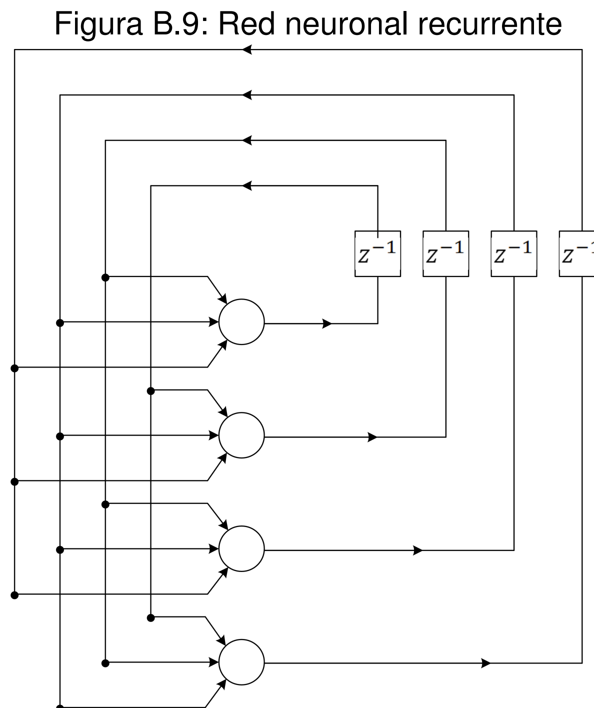
patrones de activación, los cuales así mismo constituyen la señal de entrada aplicada a las neuronas de la segunda capa, las salidas de esta capa son usadas como entrada a la tercera capa, y así sucesivamente para el resto de la red. Típicamente las neuronas tiene como señal de entrada las salidas de la capa anterior. El conjunto de salidas de las neuronas de la capa final o de salida, son así mismas la respuesta global de toda la red al patrón de activación suministrado por unidades de la capa de entrada.

Redes neuronales recurrentes

Una red neuronal recurrente se reconoce fácilmente por su forma, ya que tiene una pequeña retro-propagación, por ejemplo una red recurrente puede consistir de una sola capa de neuronas, en la que cada salida de ellas alimenta las entradas de cada neurona, algunas de estas redes contiene *self-feedback* que se refiere cuando la salida de la neurona es retroalimentada a su propia entrada. La presencia del circuito de retroalimentación tiene un gran impacto en la capacidad de aprendizaje de la red

y en su rendimiento, además el circuito de retroalimentación implica ramificaciones particulares, las cuales dan como resultado un comportamiento no lineal suponiendo que la red tiene unidades no lineales.

Figura B.9: Red neuronal recurrente



Fuente: Autores

Procesos de aprendizaje

Esta es una propiedad muy importante para una red neuronal, ya que de esta depende la capacidad de aprender del ambiente y mejorar su rendimiento a través del proceso de aprendizaje. Un red neuronal aprende de su ambiente a través de procesos iterativos que ajustan los pesos sinápticos y las bias. Idealmente, la red empezara a adquirir mas conocimiento con cada iteración en el proceso de aprendizaje.

Definición de aprendizaje

El aprendizaje es un proceso por medio del cual los parámetros libres de una red neuronal son adaptados a través de procesos de simulación del ambiente en el que la red esta sometida. El tipo de aprendizaje es determinado por la manera en que los parámetros cambian (Mendel y McClaren 1970). Esta definición de proceso de aprendizaje implica las siguiente secuencia de eventos:

- La red neuronal es estimulado por el ambiente.

- La red sufre cambios en sus parámetros libres debido a la estimulación.
- La red responde al ambiente de una forma distinta debido a los cambios que ha sufrido en la estructura interna.

Un conjunto de reglas preestablecidas y bien definidas para solucionar el proceso de aprendizaje es llamado *algoritmo de aprendizaje*. Existen muchos algoritmos de aprendizaje para el diseño de las redes neuronales, en el que cada uno de ellos ofrece sus propias ventajas. Básicamente los algoritmos difieren el uno del otro en la forma como ajustan los pesos sinápticos de la red, otro factor muy importante es la manera de como están constituidas las interconexiones de las neuronas.

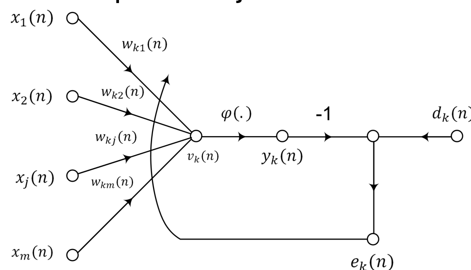
Aprendizaje de corrección del error

Para explicar este concepto, considere una red alimentada hacia adelante constituida por una unidad de procesamiento en la capa de salida como muestra la figura B.10, a este neurona le llegan conexiones de una o mas unidades escondidas de la capa oculta, las cuales así mismo se le aplica un vector de entrada proveniente de las neuronas fuentes de la capa de entrada. El argumento *n* indica el numero de la iteración en el proceso de ajuste de los pesos sinápticos de la red, la señal de salida es denotada como $y_k(n)$. Esta señal de salida representa la respuesta de la red, por lo tanto es comparada con la respuesta deseada $d_k(n)$. Por la tanto la señal de error puede definirse como:

$$e_k(n) = d_k(n) - e_k(n)$$

La señal de error $e_k(n)$ actúa como un mecanismo de control, cuyo propósito es aplicar una secuencia de ajustes correctivos a los pesos sinápticos de la neurona *k*. El ajuste correctivo esta diseñado de tal manera que la señal de salida $y_k(n)$ sea el mismo valor o aproximado a la respuesta deseada $d_k(n)$. El objetivo es minimizar la función de costo, definida en función del error como:

Figura B.10: Aprendizaje de correción del error



Fuente: Autores

$$\xi(n) = \frac{1}{2}e_k^2$$

$\xi(n)$ es un valor instantáneo de la energía del error, el proceso de ajuste de los pesos continua hasta que estos se estabilizan, es decir, hasta que la variación de los pesos con cada iteración no cambia en un valor significativo. En particular la disminución de la función de costo guía la regla de aprendizaje comúnmente llamada como *Regla delta* o *Regla de Widrow-Hoff*, llamada así en honor a sus creadores. Sí $w_{kj}(n)$ es el peso sináptico de la neurona *k* excitada por el elemento $x_j(n)$ del vector de señal $x(n)$ en una iteración *n*. Siguiendo la regla delta, el ajuste del peso $\Delta w_{kj}(n)$ aplicado a los pesos sinápticos w_{kj} en la iteración *n* se define como:

$$\Delta w_{kj}(n) = \eta e_k(n) x_j(n)$$

donde η es una constante positiva que determina la rata de aprendizaje con la que se va a proceder de una iteración a otra en el proceso de aprendizaje, en otras palabras la regla delta puede ser definida como: *El ajuste hecho a los pesos sinápticos de una neurona es proporcional a el producto de la señal de error y la señal de entrada de la sinapsis en cuestión.*

Teniendo calculado el ajuste del peso sináptico Δw_{kj}, la actualización del peso se da de la forma:

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

donde los valores $w_{kj}(n)$ y $w_{kj}(n+1)$ pueden ser vistos como el valor anterior y el nuevo del peso sináptico respectivamente.

Aprendizaje de Hebbian

El aprendizaje de Hebbian es el más viejo y famoso de las reglas de aprendizaje, es llamado así en honor al neuropsicólogo HEBB. Cuando el axón de una célula A esta lo suficientemente cerca para excitar a la célula B, y se acerca en repetidas ocasiones, los cambios metabólicos toman lugar en una o ambas células, de modo que tanto la eficiencia de la célula A como la B se incrementan. HEBB propone estos cambios como base del proceso de aprendizaje a nivel celular, debido a esto de pueden deducir dos reglas:

- Si dos neuronas a cada la de la conexión o sinapsis son activadas simultáneamente, la fuerza del peso sináptico se incrementa.
- Si dos neuronas a cada lado de la conexión son activadas asíncronamente, la sinapsis es debilitada o eliminada.

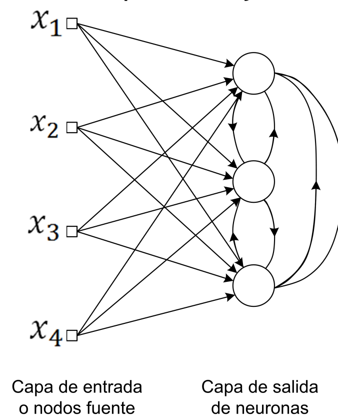
Las modificaciones del modelo matemático del aprendizaje de Hebb, considera un peso sináptico w_{kj} de una neurona *k* con señales de pre-sinapsis y post-sinapsis denotadas como x_y y y_k respectivamente, entonces el ajuste aplicado a los pesos sinápticos en la iteración *n* es expresado en forma general como:

$$\Delta w_{kj}(n) = F(y_k(n), x_j(n))$$

Aprendizaje competitivo

El aprendizaje competitivo como su nombre lo dice, es aquel donde las neuronas de salidas compiten entre ellas para llegar a ser activadas. En las redes neuronales basadas en el aprendizaje de Hebb pueden haber varias neuronas de salidas que se activan al mismo tiempo, en el aprendizaje competitivo una sola neurona puede ser activada en algún tiempo. Esta característica hace que este aprendizaje sea usada en la clasificación de patrones de datos.

Figura B.11: Aprendizaje competitivo



Fuente: Autores

Hay tres elementos básicos en las reglas de aprendizaje competitivo.

- Un conjunto de neuronas que son parecidas excepto algunas que tienen una distribución aleatoria de los pesos y que responden de manera diferente para un patrón de entrada dado.
- Un limite impuesto a la fuerza de cada neurona.
- Un mecanismo que permite a las neuronas competir por la respuesta correcta a un subconjunto de entrada dado, y en la que una y solo una neurona sera activada por grupo.

La forma más simple de aprendizaje competitivo, se da en la arquitectura con una sola capa de neuronas de salida, y en la que cada una de ellas esta completamente conectada a las neuronas de entrada y la red puede incluir conexiones *feedback* como muestra la figura B.11. Para una neurona *k* que sera la neurona ganadora, su campo de inducción v_k para un patrón de entrada x sera el más grande de todas las neuronas, la señal de salida y_k de la neurona ganadora es iguala 1, mientras que la salida de todas las otras neuronas perdedoras sera 0.

$$y_k = \begin{cases} 1 & \text{if } v_k \geq v_j \\ 0 & \text{otro caso} \end{cases}$$

Los pesos sinápticos de las conexiones de una neuronas esta dados de tal forma que la sumatoria de cada uno de ellos de 1.

$$\sum_{j} w_{kj} = 1 \text{ para cada neurona k}$$

El ajuste de los pesos sinápticos Δw_{kj} para el aprendizaje competitivo esta dado por:

$$\Delta w_{kj} = \begin{cases} \eta(x_j - w_{kj}) & \text{si la neurona es ganadora} \\ 0 & \text{si la neurona es perdedora} \end{cases}$$

donde η es la rata de aprendizaje.

Aprendizaje de Boltzmann

Las reglas de aprendizaje de Boltzmann, es un algoritmo de aprendizaje estocástico derivadas de las ideas de la mecánica estadística. Una red neuronal diseñada tomando como base las reglas de aprendizaje de Boltzmann es llamada *maquina de Boltzmann*. Las neuronas de una maquina de Boltzmann constituyen una estructura recurrente, y operan de una manera binaria, por ejemplo, si la neurona esta activada entonces su estado es denotado por +1, o si esta desactivada su estado es denotado por -1. La maquina de Boltzmann es caracterizada por la *Función de Energia*, E, cuyo valor es determinado por estado particular de las neuronas de la maquina y se calcula de la siguiente manera:

$$E = -\frac{1}{2}\sum_{j}\sum_{k} w_{kj}x_k x_j$$

donde x_j es el estado de la neurona j, w_{kj} es el peso sináptico de la conexión de la neurona j con la neurona k. La maquina opera escogiendo una neurona aleatoria, por ejemplo una neurona k, en una iteración del proceso de aprendizaje, el estado de la neurona k va desde x_k a $-x_k$ a una temperatura T con la probabilidad:

$$P(x_k > -x_k) = \frac{1}{1 + exp(-\Delta E_k/T)}$$

Las neuronas de la máquina de Boltzmann se dividen en dos grandes grupos: visibles y escondidas, las visibles proporcionan una interfaz entre la red y el ambiente, y las neuronas escondidas operan de manera libre. Hay dos formas de operar:

- Condición enlazada: en la que las neuronas visibles están enlazadas por un estado especifico determinado por el ambiente.
- Operación libre: en el que todas las neuronas se les permite operar de forma libre.

Si ρ_{kj}^{+} es la correlación entre el estado de la neurona j y k, con la red en condición de operación enlazada, y ρ_{kj}^{-} es la correlación entre estado de la neurona j y k, con la condición de operación libre, estas correlaciones son promediadas para todos los estados posibles de la maquina en el estado de equilibrio térmico. Entonces acordado con la regla de aprendizaje de Boltzmann, el ajuste del peso Δw_{kj} aplicado al peso sinápticos de la neurona j a la k es definido como:

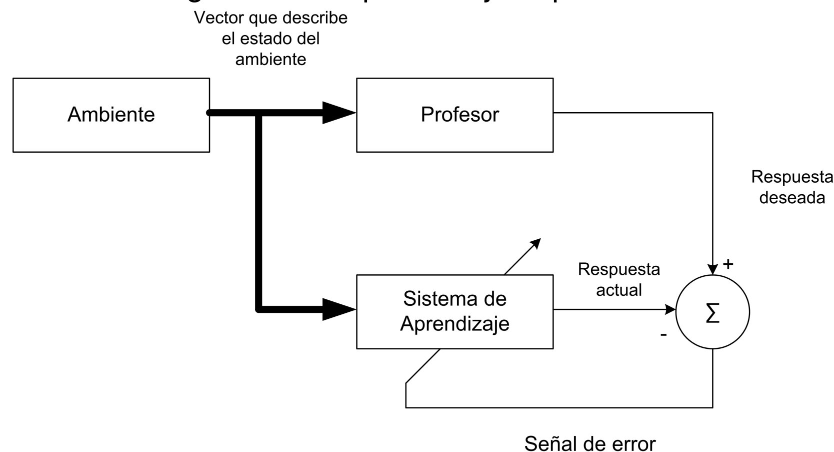
$$\Delta w_{kj} = \eta(\rho_{kj}^{+} - \rho_{kj}^{-}) j \neq k$$

Dentro de los algoritmos de aprendizaje se encuentran diversos tipos como los siguientes:

Aprendizaje con un profesor

La figura B.12 muestra el proceso de aprendizaje con un profesor o supervisado, en términos conceptuales se puede pensar que el profesor es el que tiene el conocimiento del ambiente, y este conocimiento esta representado como un conjunto de entradas y ejemplos de salida, el ambiente es sin embargo desconocido para la red de interés.

Figura B.12: Aprendizaje supervisado



Fuente: Autores

Supongamos que el profesor y la red son expuestos a un vector de entrenamiento proporcionado por el ambiente, por virtud del conocimiento adquirido el profesor es capaz de proporcionar a la red una respuesta deseada para el vector de entrenamiento, esta respuesta deseada es la acción óptima de la red. Los parámetros de la red son ajustados bajo la influencia del vector de entrenamiento y la señal de error, donde la señal de error es definida como la diferencia entre la respuesta deseada y la respuesta actual de la red. Este ajuste es llevado en un proceso iterativo donde el objetivo es que la red emule al profesor, esta emulación se supone que es el estado

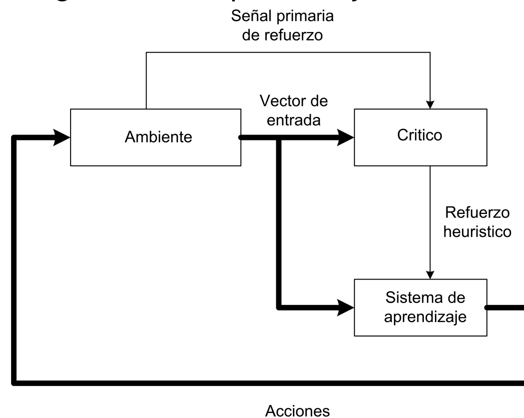
óptimo en el sentido estadístico. En este camino, el conocimiento en el ambiente disponible al profesor es transferido en la mayor cantidad posible a la red neuronal. Cuando la condición óptima es alcanzada, se puede prescindir del profesor y dejar que la red interactué con el ambiente directamente. Para medir el rendimiento del sistema se define una función de los parámetros libres del sistema, esta función puede ser vista como un error de rendimiento superficial o error de superficie, este error es un promedio global de las posibles entradas y ejemplos de salida. En una operación dada del sistema bajo la supervisión del profesor, es representada como un punto en el error de superficie, para mejorar el rendimiento el punto de operación tiene que bajar hacia un punto donde el error de superficie es mínimo.

Aprendizaje sin profesor

El aprendizaje supervisado se da bajo la tutoría de un profesor, sin embargo en el paradigma conocido como *aprendizaje sin profesor* como su nombre lo dice, no hay un profesor que supervise el proceso de aprendizaje, bajo este segundo paradigma existen dos clasificaciones:

- Aprendizaje reforzado: En el aprendizaje reforzado la asignación de entradas y salidas es realizado a través de continuas interacciones con el ambiente. La figura B.13 muestra un diagrama de bloques de una forma de sistema de aprendizaje reforzado construido alrededor de un elemento critico que convier-

Figura B.13: Aprendizaje reforzado



Fuente: Autores

  te una señal primaria reforzada recibida del ambiente a una señal de calidad más alta llamada *señal heurística de refuerzo*, donde ambas son escalares.

  El sistema es diseñado para aprender bajo el concepto de *refuerzo retardado*, el cual significa que el sistema observa una secuencia temporal de estímulos, también recibidos del ambiente, y con las cuales resulta la generación de señales heurísticas reforzadas. El objetivo del aprendizaje es minimizar la función

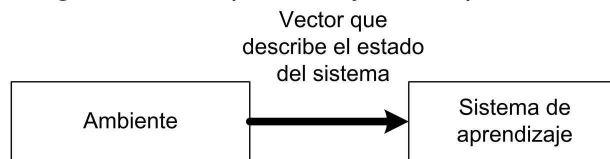
de costo, definida como la acumulación de costo de las acciones asumidas en cada paso de la iteración, en vez de un simple costo inmediato. Esto podría producir que algunas acciones tomadas en la iteración tempranamente sean las mejores de todo el sistema, la función de la maquina de aprendizaje, que constituye el segundo componente del sistema, es descubrir las acciones y alimentarlas de nuevo al ambiente.

El aprendizaje reforzado en retraso es difícil de mejorar debido a dos aspectos básicos:

- No hay un profesor que suministre la respuesta deseada en cada paso del proceso de aprendizaje.
- El retraso incurre en la generación de señales primarias reforzadas y esto implica que la maquina de aprendizaje debe solucionar el problema de asignación de crédito temporal.

• Aprendizaje no supervisado: En este aprendizaje no hay un profesor externo o critico que supervise el proceso de aprendizaje, como indica la figura B.14. El suministro es hecho por un medidor de tareas independiente de la calidad de la representación que la red requiere para aprender, y los parámetros libres de la red son optimizados por el medidor. Una vez que la red a llegado a afinar

Figura B.14: Aprendizaje no supervisado



Fuente: Autores

las estadísticas regulares de la entrada de datos, esta desarrolla la habilidad de formar representaciones internas para codificar las características de la entrada y de este modo crear nuevas clases automáticamente. Para mejorar el aprendizaje no supervisado se debe usar una regla de aprendizaje competitiva, por ejemplo, se puede usar una red que consiste de dos capas, una capa de entrada y una capa competitiva, la capa de entrada recibe los datos disponibles, y la capa competitiva consiste de neuronas que compiten entre ellas por la oportunidad de responder a la entrada de datos.

C. ARCHIVO RESPUESTA DE LAS LIBRERIAS

```
PROJECT Iris

%definición de las variables en forma de conuntos fuzzy

    VAR Longitud de sepalo        %nombre de la variable
    TYPE float                    %tipos de valores
    MIN 4                         %valor minimo de la variable
    MAX 8                         %valor máximo de la variable

        MEMBER pequeno            %Etiqueta del conjunto fuzzy
            POINTS 4,1 6.15523,1 7.99442,0  %puntos del conjunto fuzzy
        END
        MEMBER mediano
            POINTS 4.00108,0 6.15559,1 7.99999,0
        END
        MEMBER grande
            POINTS 4.00107,0 6.15862,1 8,1
        END
    END

    VAR Ancho de sepalo
    TYPE float
    MIN 2
    MAX 5
        MEMBER pequeno
            POINTS 2,1 3.59277,1 5,0
        END
        MEMBER mediano
            POINTS 2.00317,0 3.59307,1 5,0
        END
        MEMBER grande
            POINTS 2.23578,0 3.61789,1 5,1
        END
    END

    VAR Longitud del petalo
    TYPE float
    MIN 1
    MAX 7
        MEMBER pequeno
            POINTS 1,1 2.15059,1 3.30118,0
        END
        MEMBER mediano
            POINTS 1.54918,0 3.98338,1 6.41758,0
```

```
        END
        MEMBER grande
            POINTS 3.71924,0 5.35962,1 7,1
        END
    END

    VAR Ancho del petalo
    TYPE float
    MIN 0
    MAX 3
        MEMBER pequeno
            POINTS 0,0 1.14293,1 2.28585,0
        END
        MEMBER mediano
            POINTS 0.148305,0 1.17323,1 1.9683,0
        END
        MEMBER grande
            POINTS 1.40896,0 2.20448,1 3,1
        END
    END

%definición de las reglas fuzzy del sistema

RULEBASE Reglas
    RULE Rule1 IF (Longitud de sepalo IS pequeno) AND (Ancho de sepalo IS
        mediano) AND (Longitud del petalo    IS pequeno) AND (Ancho del
       petalo IS pequeno) THEN Current = Iris Setosa
    END

    RULE Rule2 IF (Longitud de sepalo IS grande) AND (Ancho de sepalo IS
       mediano) AND (Longitud del petalo  IS mediano) AND (Ancho del
       petalo IS mediano) THEN Current = Iris Versicoulor
    END

    RULE Rule3 IF (Longitud de sepalo IS mediano) AND (Ancho de sepalo IS
        mediano) AND (Longitud del petalo    IS grande) AND (Ancho del
       petalo IS grande) THEN Current = Iris Virginica
    END

    RULE Rule4 IF (Longitud de sepalo IS pequeno) AND (Ancho de sepalo IS
        pequeno) AND (Longitud del petalo    IS pequeno) AND (Ancho del
       petalo IS pequeno) THEN Current = Iris Setosa
    END

    RULE Rule5 IF (Longitud de sepalo IS mediano) AND (Ancho de sepalo IS
        mediano) AND (Longitud del petalo    IS mediano) AND (Ancho del
       petalo IS mediano) THEN Current = Iris Versicoulor
    END

    RULE Rule6 IF (Longitud de sepalo IS mediano) AND (Ancho de sepalo IS
        pequeno) AND (Longitud del petalo    IS grande) AND (Ancho del
       petalo IS grande) THEN Current = Iris Virginica
```

END

RULE Rule7 **IF** (Longitud de sepalo **IS** grande) **AND** (Ancho de sepalo **IS** pequeno) **AND** (Longitud del petalo **IS** grande) **AND** (Ancho del petalo **IS** mediano) **THEN** Current = Iris Virginica
END

RULE Rule8 **IF** (Longitud de sepalo **IS** grande) **AND** (Ancho de sepalo **IS** pequeno) **AND** (Longitud del petalo **IS** grande) **AND** (Ancho del petalo **IS** grande) **THEN** Current = Iris Virginica
END

RULE Rule9 **IF** (Longitud de sepalo **IS** pequeno) **AND** (Ancho de sepalo **IS** pequeno) **AND** (Longitud del petalo **IS** mediano) **AND** (Ancho del petalo **IS** mediano) **THEN** Current = Iris Versicoulor
END

RULE Rule10 **IF** (Longitud de sepalo **IS** mediano) **AND** (Ancho de sepalo **IS** pequeno) **AND** (Longitud del petalo **IS** grande) **AND** (Ancho del petalo **IS** mediano) **THEN** Current = Iris Versicoulor
END

RULE Rule11 **IF** (Longitud de sepalo **IS** mediano) **AND** (Ancho de sepalo **IS** pequeno) **AND** (Longitud del petalo **IS** mediano) **AND** (Ancho del petalo **IS** mediano) **THEN** Current = Iris Versicoulor
END

RULE Rule12 **IF** (Longitud de sepalo **IS** pequeno) **AND** (Ancho de sepalo **IS** grande) **AND** (Longitud del petalo **IS** pequeno) **AND** (Ancho del petalo **IS** pequeno) **THEN** Current = Iris Setosa
END

RULE Rule13 **IF** (Longitud de sepalo **IS** pequeno) **AND** (Ancho de sepalo **IS** pequeno) **AND** (Longitud del petalo **IS** mediano) **AND** (Ancho del petalo **IS** pequeno) **THEN** Current = Iris Versicoulor
END

RULE Rule14 **IF** (Longitud de sepalo **IS** grande) **AND** (Ancho de sepalo **IS** pequeno) **AND** (Longitud del petalo **IS** mediano) **AND** (Ancho del petalo **IS** mediano) **THEN** Current = Iris Versicoulor
END

RULE Rule15 **IF** (Longitud de sepalo **IS** grande) **AND** (Ancho de sepalo **IS** mediano) **AND** (Longitud del petalo **IS** grande) **AND** (Ancho del petalo **IS** grande) **THEN** Current = Iris Virginica
END

RULE Rule16 **IF** (Longitud de sepalo **IS** mediano) **AND** (Ancho de sepalo **IS** pequeno) **AND** (Longitud del petalo **IS** mediano) **AND** (Ancho del petalo **IS** pequeno) **THEN** Current = Iris Versicoulor
END

```
RULE Rule17 IF (Longitud de sepalo IS mediano) AND (Ancho de sepalo
   IS grande) AND (Longitud del petalo    IS pequeno) AND (Ancho del
    petalo IS pequeno) THEN Current = Iris Setosa
END

RULE Rule18 IF (Longitud de sepalo IS mediano) AND (Ancho de sepalo
   IS mediano) AND (Longitud del petalo   IS pequeno) AND (Ancho del
    petalo IS pequeno) THEN Current = Iris Setosa
END

RULE Rule19 IF (Longitud de sepalo IS mediano) AND (Ancho de sepalo
   IS mediano) AND (Longitud del petalo   IS grande) AND (Ancho del
   petalo IS mediano) THEN Current = Iris Versicoulor
END
```

```
%Resultado de la validación
la validacion para este entrenamiento fue del 97%

%Resultado de la clasificación
El patron numero 1 es clase 1
El patron numero 2 es clase 2
El patron numero 3 es clase 3
El patron numero 4 es clase 1
El patron numero 5 es clase 2
El patron numero 6 es clase 3
El patron numero 7 es clase 1
El patron numero 8 es clase 2
El patron numero 9 es clase 3
El patron numero 10 es clase 1
El patron numero 11 es clase 2
El patron numero 12 es clase 3
El patron numero 13 es clase 1
El patron numero 14 es clase 2
El patron numero 15 es clase 3
El patron numero 16 es clase 1
El patron numero 17 es clase 2
El patron numero 18 es clase 3
El patron numero 19 es clase 1
El patron numero 20 es clase 2
El patron numero 21 es clase 3
El patron numero 22 es clase 1
El patron numero 23 es clase 2
El patron numero 24 es clase 3
El patron numero 25 es clase 1
El patron numero 26 es clase 2
El patron numero 27 es clase 2
El patron numero 28 es clase 1
El patron numero 29 es clase 2
El patron numero 30 es clase 2
El patron numero 31 es clase 1
El patron numero 32 es clase 2
El patron numero 33 es clase 3
```

El patron numero 34 es clase 1
El patron numero 35 es clase 2
El patron numero 36 es clase 3
El patron numero 37 es clase 1
El patron numero 38 es clase 2
El patron numero 39 es clase 3
El patron numero 40 es clase 1
El patron numero 41 es clase 2
El patron numero 42 es clase 3
El patron numero 43 es clase 1
El patron numero 44 es clase 2
El patron numero 45 es clase 3
El patron numero 46 es clase 1
El patron numero 47 es clase 2
El patron numero 48 es clase 3
El patron numero 49 es clase 1
El patron numero 50 es clase 2
El patron numero 51 es clase 3
El patron numero 52 es clase 1
El patron numero 53 es clase 2
El patron numero 54 es clase 3
El patron numero 55 es clase 1
El patron numero 56 es clase 2
El patron numero 57 es clase 3
El patron numero 58 es clase 1
El patron numero 59 es clase 2
El patron numero 60 es clase 3
El patron numero 61 es clase 1
El patron numero 62 es clase 2
El patron numero 63 es clase 3
El patron numero 64 es clase 1
El patron numero 65 es clase 2
El patron numero 66 es clase 3
El patron numero 67 es clase 1
El patron numero 68 es clase 2
El patron numero 69 es clase 3
El patron numero 70 es clase 1
El patron numero 71 es clase 2
El patron numero 72 es clase 3
El patron numero 73 es clase 1
El patron numero 74 es clase 2
El patron numero 75 es clase 3