

Resumen Final Inteligencia Computacional

Cristian Escudero
Con agradecimientos a Marcos Yedro

24 de julio de 2013

Índice

I	Introducción	6
1.	¿Qué es la Inteligencia Computacional?	6
2.	¿Qué son las Redes Neuronales?	6
2.1.	Beneficios de las RNA	6
2.2.	Cerebro Humano	7
2.3.	Modelos de Neuronas	8
2.4.	Arquitecturas de RNAs	9
3.	Procesos de Aprendizaje	10
3.1.	Nociones Básicas	10
3.2.	Algoritmos de Aprendizaje	10
3.2.1.	Por Corrección de Error	10
3.2.2.	Basado en Memoria	10
3.2.3.	Boltzmann	10
3.2.4.	Hebbiano	11
3.2.5.	Competitivo	11
3.3.	Paradigmas de Aprendizaje	11
3.4.	Tareas de aprendizaje	11
3.4.1.	Aplicaciones de las RNAs	12
4.	Perceptrón Simple	12
4.1.	Algoritmo de aprendizaje	12
4.2.	Entrenamiento por método de gradiente	12
4.2.1.	Algoritmo de Least-Mean-Square	13
4.2.2.	Ejemplo gráfico de resolución del problema OR	13
5.	Bases estadísticas de reconocimiento de patrones	13
5.1.	Definiciones	13
5.2.	Aproximaciones al reconocimiento de patrones	14
5.2.1.	Aproximación Estadística	14
5.2.2.	Aproximación Sintáctica	14
5.3.	Paradigma de trabajo funcional:	14
5.4.	Clasificador estadístico	15
5.4.1.	Tipos de probabilidades	15
5.4.2.	Clasificador de Bayes	15
6.	Perceptrón Multicapa	15
6.1.	Introducción	15
6.2.	Algoritmo de retropropagación	16
6.2.1.	Cálculo de las salidas en cada capa	16
6.2.2.	Criterio de Error	17
6.2.3.	Resumen del algoritmo de retropropagación	19
6.3.	Tasa de Aprendizaje: Constante de Momento	19
6.4.	Algunas heurísticas para mejorar la <i>performance</i>	19
6.5.	Generalización	20
6.6.	Validación Cruzada	20
6.6.1.	Método de Detención Temprana de Entrenamiento	20
6.6.2.	Características	20
6.6.3.	Variantes de la Validación Cruzada	21
7.	Red Neuronal de función de Base Radial	21
7.1.	Introducción	21
7.1.1.	Construcción	21
7.2.	Entrenamiento en la Capa RBF	22

7.2.1.	Algoritmos de adaptación	22
7.3.	Entrenamiento en la tercera capa	23
7.3.1.	Entrenamiento de pesos usando LMS	23
7.3.2.	Entrenamiento de pesos usando la pseudo-inversa del vector $\phi(\mathbf{x}_l)$	23
7.4.	Comparación entre las RBF y los MLP	23
8.	Redes Competitivas	24
8.1.	Mapas Auto-Organizativos	24
8.1.1.	Dos modelos básicos de mapeo	24
8.1.2.	Algoritmo de formación	24
8.2.	Learning Vector Quantifization (LVQ)	26
8.2.1.	Algoritmo de aprendizaje básico	26
8.2.2.	Diferencias con el SOM	27
9.	Redes Neuronales Dinámicas	27
9.1.	Introducción	27
9.2.	Clasificación	27
9.3.	Red de Hopfield	28
9.3.1.	Características	28
9.3.2.	Fases de Operación	28
9.3.3.	Estados espurios	28
9.3.4.	Capacidad de almacenamiento	29
9.3.5.	Generalidades	29
9.4.	Arquitecturas de RR	29
9.4.1.	Modelo NARX	29
9.4.2.	Modelo de Espacio de Estados	29
9.4.3.	MLP recurrente (RMLP)	30
9.4.4.	Redes de Segundo Orden	30
9.5.	Redes Dinámicas de Elman y Jordan	30
9.6.	Retropropagación a través del tiempo	30
II	Lógica borrosa	32
1.	Lógica	32
1.1.	Lógica Proposicional	32
1.1.1.	Reglas de Inferencia	32
1.2.	Lógica de Predicados de Primer Orden	33
1.2.1.	Reglas de Inferencia	33
1.3.	Sistemas de producción con encadenamiento hacia adelante	33
1.3.1.	Componentes:	33
1.3.2.	Fases:	34
2.	Lógica Borrosa	34
2.1.	Introducción	34
2.1.1.	Motivación	34
2.1.2.	Conjuntos borrosos y binarios	35
2.2.	Operaciones básicas	35
2.2.1.	Operaciones elementales	36
2.2.2.	Distancias borrosas	36
2.3.	Caracterización de los conjuntos borrosos	37
2.3.1.	Conjunto binario de nivel α	37
2.3.2.	Conjunto convexo	37
2.3.3.	Conjunto normal	37
2.3.4.	Cardinalidad de un conjunto	37
2.3.5.	Propiedades a tener en cuenta	37
2.3.6.	El conjunto borroso medio	38
2.3.7.	Entropía borrosa	38
2.3.8.	Teorema de la entropía borrosa	38

2.3.9. Teorema del subconjunto borroso	39
2.3.10. Teorema de la entropía y el subconjunto borroso	39
3. Memorias Asociativas Borrosas	39
3.1. Introducción	39
3.2. Fuzzificación	39
3.3. Codificación	40
3.3.1. Multiplicación borrosa de Vector-Matrix: Composición máx-mín	40
3.3.2. Fuzzy Hebbs FAMs	40
3.3.3. Teorema de Bidireccionalidad de la correlación-mínimo	41
3.3.4. Algunas consideraciones prácticas	41
3.4. Composición	41
3.4.1. Sobreposición de reglas FAM	41
3.5. Defuzzificación	42
3.6. Conjuntos de pertenencia continuos	42
3.6.1. Cálculo de defuzzificación para los centroides trapezoidales	42
3.6.2. Múltiples antecedentes por consecuente	42
3.7. Otras alternativas	43
3.8. Comparación entre NN vs FS vs ES	43
3.8.1. Sobre Mamdani, Takagi-Sugeno-Kang y SAM	44
III Inteligencia Colectiva	45
1. Resolver problemas mediante búsqueda	45
1.1. Nociones Básicas	45
1.1.1. Definición de agente	45
1.1.2. Diferencia entre agente y programa/objeto	45
1.1.3. Definición de agente racional	45
1.1.4. Clasificación	45
1.2. Agentes resolvedores de problemas	46
1.3. Problemas y soluciones bien definidos	46
1.3.1. Árbol de Búsqueda	46
1.4. Medir el rendimiento de la resolución del problema	47
2. Planificación	48
2.1. Lenguaje de los problemas de planificación	48
2.2. Operadores STRIPS	48
2.3. Planificador de Orden Parcial	49
3. Computación Evolutiva	49
3.1. Algoritmos genéticos	50
3.2. La evolución como un algoritmo	50
3.3. Elementos de un algoritmo evolutivo	50
3.4. Diseño de una solución mediante algoritmos evolutivos	51
3.4.1. Representación de los individuos	51
3.4.2. Estrategias de selección	51
3.4.3. Operadores de variación y reproducción	51
3.5. Características principales y variantes	52
3.5.1. Características principales	52
3.5.2. Tratamiento de las restricciones del problema	52
3.6. Programación Genética	53
4. Enjambres de partículas y colonias de hormigas	53
4.1. Autómatas de estados finitos y autómatas celulares	53
4.2. Optimización por Enjambre de Partículas	54
4.2.1. Inspiración biológica de los métodos de inteligencia colectiva	54
4.2.2. Estructura social	54
4.2.3. Algoritmo	54

4.2.4.	Convergencia	56
4.2.5.	Parámetros del Sistema	56
4.2.6.	Comparación con los algoritmos genéticos	56
4.3.	Colonias de Hormigas	56
4.3.1.	Inspiración biológica	56
4.3.2.	Feromonas	57
4.3.3.	Algoritmo básico	57

Parte I

Introducción

1. ¿Qué es la Inteligencia Computacional?

La **Inteligencia Computacional** (IC) es una rama de la **inteligencia artificial** (IA), centrada en el estudio de **mecanismos adaptativos** para permitir el comportamiento inteligente de sistemas complejos y cambiantes. Una alternativa a la *IA clásica*, que trata de no confiar en algoritmos heurísticos tan habituales en esta última.

La investigación en ésta área no rechaza los métodos estadísticos, incorporando a menudo una vista complementaria. Dentro de la IC podemos encontrar técnicas como las *Redes Neuronales*, *Computación Evolutiva*, *Inteligencia Colectiva*, y *Lógica Borrosa*.

2. ¿Qué son las Redes Neuronales?

El trabajo en las **redes neuronales artificiales** (RNA) ha sido motivado desde sus inicios por el reconocimiento de que el cerebro humano funciona computacionalmente de una forma totalmente diferente al de las computadoras convencionales digitales: el *cerebro* es altamente: *complejo*, *no-lineal*, y además *procesa en paralelo*. Tiene la capacidad de organizar sus componentes estructurales (*neuronas*) de forma tal de realizar ciertas operaciones computacionales¹ de manera mucho más rápida que las computadoras actuales.

El cerebro tiene la habilidad de construir sus propias reglas en base a la “*experiencia*”. Esta *plasticidad* permite al sistema nervioso en desarrollo adaptarse al ambiente que lo rodea.

En su forma más general, una RNA es una máquina diseñada para *modelar* la forma en que el cerebro resuelve una cierta tarea en particular de interés; su implementación luego se logra mediante SW o HW.

Definición:

Una RNA es un procesador masivo paralelo y distribuido, construido a partir de simples unidades de procesamiento que tienen la capacidad natural de almacenar conocimiento y localizarlo para su uso. Se asemeja al cerebro humano en dos aspectos:

1. El conocimiento es adquirido por la red desde el ambiente a través de un proceso de **aprendizaje**.
2. El conocimiento adquirido es almacenado mediante los **pesos sinápticos** de cada conexión de neuronas.

2.1. Beneficios de las RNA

Se derivan principalmente de la habilidad de aprender y *generalizar*², y del procesamiento masivo distribuido en paralelo. El uso de las RNA brinda las siguientes propiedades y capacidades:

1. **Aprendizaje**. Las RNA tienen la habilidad de aprender mediante una etapa llamada *etapa de aprendizaje*.
2. **Adaptabilidad (*plasticidad*)**. Las RNA poseen la capacidad de adaptar sus pesos según los cambios que se produzcan en el ambiente que la rodea. Relacionado a esto está el dilema de *estabilidad-plasticidad*³.

¹Por ejemplo, reconocimiento de patrones, percepción, control motor.

²La *generalización* se refiere a la RNA produciendo salidas razonables a partir de entradas que no se hallaban en el conjunto de entrenamiento.

³*Stability-Plasticity Dilemma*: un sistema de aprendizaje es *estable* si ningún patrón dentro del conjunto de entrenamiento cambia de categoría luego de un número finito de iteraciones. Esto puede lograrse forzando a la tasa de aprendizaje (η) a decrecer gradualmente a cero. Pero esto causa otro problema conocido como *plasticibilidad*, que es la habilidad de adaptarse a la nueva información.

3. **Auto-organización.** Una RNA crea su propia representación de la información en su interior, liberando al usuario de esto. Es decir, forma su propio mapeo de entrada/salida.
4. **Información contextual.** Cada neurona puede ser influenciada por la actividad de las otras neuronas de la red.
5. **No-linealidad.** Una RNA puede ser tanto lineal como no-lineal. La no-linealidad es una propiedad importante cuando el problema subyacente es inherentemente no-lineal. Por ejemplo, una *señal de voz*.
6. **Tiempo real.** La estructura de una RNA es paralela, por lo cual si esto es implementado con computadoras o en dispositivos electrónicos especiales, se pueden obtener respuestas en tiempo real.
7. **Tolerancia a fallos.** Debido a que una RNA almacena la información de forma redundante (distribuida), ésta puede seguir respondiendo de manera aceptable aun si se daña parcialmente.
8. **Respuesta evidencial.** La red puede indicar que grado de *confianza* tiene el resultado dado, y de esa forma detectar ambigüedades y mejorar el sistema de clasificación.
9. **Analogía neuro-biológica.** Su diseño está inspirado en analogía con el cerebro, el cuál funciona como un procesador poderoso y rápido gracias a su procesamiento en paralelo.

Sin embargo, las RNA poseen las siguientes desventajas:

- Requieren gran cantidad de datos de entrenamiento diversos para operaciones en el mundo real.
- Requieren gran tiempo de procesamiento y de memoria.

2.2. Cerebro Humano

El sistema nervioso central humano puede verse como un sistema de tres etapas (**Figura 1**). Central al sistema está el *cerebro*, que recibe continuamente información, la interpreta, y toma decisiones en base a ella. Las flechas \rightarrow indican transmisiones *feed-forward*, y las flechas \leftarrow marcan transmisiones *feed-back*.

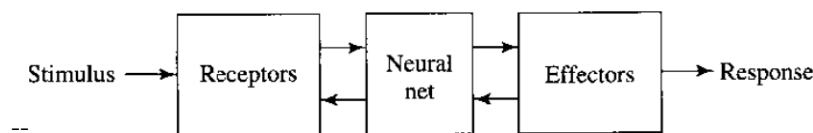


Figura 1: Diagrama en bloques representando el sistema nervioso.

Aunque la velocidad de operación de las neuronas es relativamente lenta con respecto a los circuitos digitales integrados, se compensa por la gran cantidad de neuronas (aproximadamente *10 mil millones*), y la masividad de conexiones entre ellas (cerca de *60 trillones* de sinapsis).

Las **sinapsis** son estructuras elementales y unidades funcionales que median la interacción entre las neuronas. Se asume que una sinapsis es una conexión simple que puede imponer una *excitación* o una *inhibición*, pero no ambas en la neurona receptora.

En el cerebro de un adulto, la plasticidad se logra en base a dos mecanismos:

- la creación de conexiones sinápticas entre neuronas;
- y la modificación de conexiones sinápticas existentes.

La mayoría de las neuronas codifican sus salidas como una serie de breves impulsos de voltaje (*acciones potenciales*). Esto es así debido a la física de los *axones* (las líneas de transmisión), que pueden ser modelados como circuitos RC.

En el cerebro hay tanto organizaciones anatómicas de pequeña como de gran escala, y diferentes funciones se dan a lugar tanto en los niveles altos como bajos.

Nivel Estructural	Descripción
<i>Sistema Nervioso Central</i>	Unión de <i>circuitos inter-regionales</i> y <i>mapas topográficos</i> , que establecen distintos tipos específicos de comportamiento.
<i>Circuitos Inter-regionales</i>	Conjunto de múltiples <i>circuitos locales</i> localizados en diversas partes del cerebro.
<i>Circuitos Locales</i>	Agrupación de <i>neuronas</i> para realizar operaciones características de una determinada región del cerebro.
<i>Neuronas</i>	Cada neurona posee varias <i>sub-unidades dendríticas</i> .
<i>Sub-unidades dendríticas</i>	Agrupación de <i>microcircuitos neurales</i> .
<i>Microcircuitos neurales</i>	Conjunto de <i>sinapsis</i> organizadas en patrones de conectividad para producir una operación funcional de interés.
<i>Sinapsis</i>	Nivel fundamental. Depende sólo de <i>moléculas</i> y de <i>iones</i> para su accionar.

Nota: Los *mapas topográficos* están organizados para responder información sensorial entrante.

2.3. Modelos de Neuronas

Una neurona es una **unidad de procesamiento de información fundamental** de una RNA. Su modelo posee tres elementos básicos:

1. Un conjunto de *enlaces conectores*, cada uno de los cuales posee un cierto **peso sináptico** que lo caracteriza. En una RNA, puede ser un valor positivo o negativo.
2. Un **sumador** que hace una *combinación lineal* de las señales de entrada ponderadas por los respectivos pesos sinápticos de la neurona.
3. Una **función de activación** que limita la amplitud de la salida de la neurona.

El modelo neuronal anterior incluye además un cierto *bias* (umbral). Esta variable tiene el efecto de incrementar o disminuir el umbral necesario para que la neurona se active.

El uso del *bias* b_k tiene el efecto de aplicar una *transformación afín*⁴ a la salida u_k , de forma tal que el gráfico v_k en función de u_k puede no pasar a través del origen. Matemáticamente:

$$u_k = \sum_{j=1}^m w_{kj} x_j,$$

$$v_k = u_k + b_k,$$

$$y_k = \phi(v_k).$$

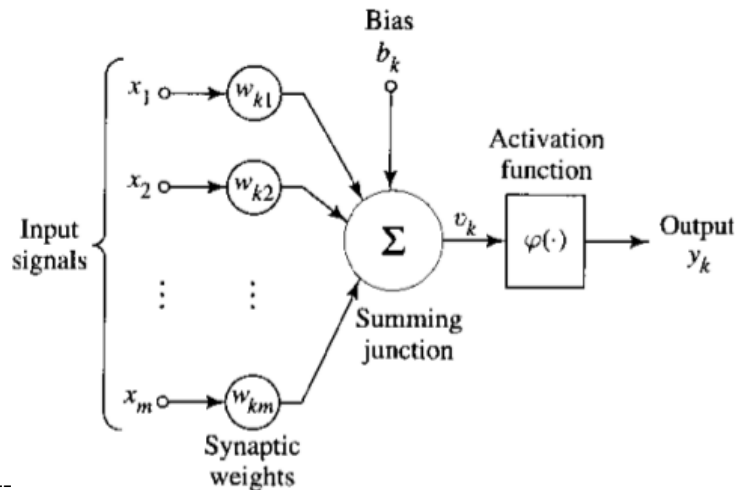


Figura 2: Modelo de una neurona no-lineal.

La función de activación $\phi(v_k)$ define la salida de una neurona en base al valor de v . Existen distintos tipos de funciones de activación, que pueden estar en el rango $[0, 1]$ o $[-1, 1]$. Algunas de ellas dentro del rango $[0, 1]$ son:

⁴Una *transformación afín* consiste en una transformación lineal seguida de una traslación: $x \rightarrow Ax + b$.

1. **Función Umbral ó Escalón:**

$$\phi(v) = \begin{cases} 1, & \text{if } v \geq 0 \\ 0, & \text{if } v < 0 \end{cases}$$

2. **Función de Activación Lineal:**

$$\phi(v) = \begin{cases} 1, & \text{if } v \geq C \\ v, & \text{if } v \in [-C, C] \\ 0, & \text{if } v \leq -C \end{cases}$$

dónde C es una constante que generalmente toma el valor de $\frac{1}{2}$.

3. **Función Sigmoidea.** Es la más común en RNA. Se define como una función estrictamente creciente que logra un balance entre

un comportamiento lineal y no-lineal. Un ejemplo de esta es la *función logística*:

$$\phi(v) = \frac{1}{1 + \exp(-av)}.$$

Donde a controla la pendiente de la función. La principal ventaja consiste en que la $\phi(v)$ es diferenciable, y además puede parecerse tanto como se quiera a la *función escalón* aumentando el valor de a . La *tangente hiperbólica* $\phi(v) = \tanh(v)$ proporciona el mismo comportamiento pero en el rango $[-1, 1]$.

A veces se desea que la función de activación sea del tipo **estocástica**. En estos casos, la decisión de una neurona de activarse es *probabilística*. También puede utilizarse la *función sigmoidea* para estos casos.

2.4. Arquitecturas de RNAs

Tenemos dos fundamentalmente diferentes clases de arquitecturas de RNAs, cuya organización está muy relacionada al **algoritmo de aprendizaje** utilizado.

- **Redes feed-forward.** Cuentan con una *capa de entrada*, una *capa de salida*, y, en el caso de las *multilayer*, con una o más *capas ocultas*. Puede estar *completamente* ó *parcialmente conectada*. Las salidas de la capa n son entradas de la capa $n+1$. Se dice que son *estáticas*, ya que producen un solo conjunto de patrones de salida más que una secuencia de patrones. Además, son *memory-less* en el sentido de que la respuesta a cierta entrada es independiente a los patrones de entrada anteriores.
- **Redes recurrentes.** Estas redes se distinguen de las redes *feed-forward* porque poseen al menos un *bucle feed-back* de retroalimentación. Pueden contener una o varias capas. Estas redes poseen un comportamiento *no-lineal dinámico*, debido a la presencia de elementos de *unit-delay* (denotados como bloques de z^{-1}).

Nota: también existen modelos **híbridos**, que pueden ser *parcialmente recurrentes* (existen reglas con respecto hacia dónde se hacen los bucles), o redes *modulares* (redes de RNAs).

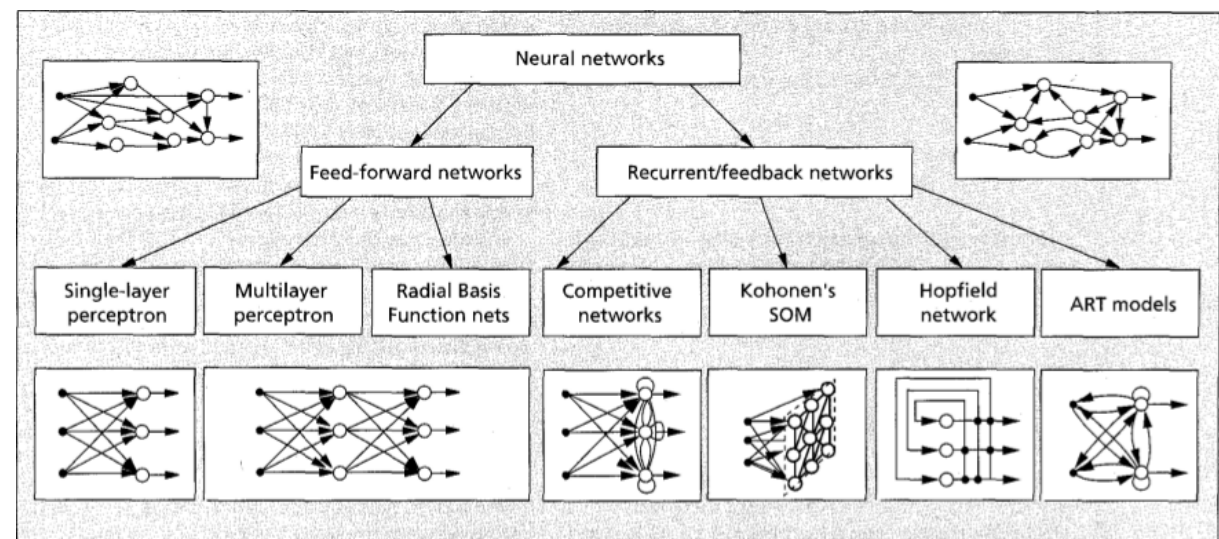


Figura 3: Clasificación de arquitecturas de RNAs.

3. Procesos de Aprendizaje

3.1. Nociones Básicas

Aprendizaje: proceso por el cual los parámetros libres de una RNA son adaptados a través de un proceso de estimulación provocado por el entorno en el cual la red se encuentra. El **tipo de aprendizaje** es determinado por la manera en la cual se realiza el cambio de parámetros.

Algoritmo de aprendizaje: conjunto de reglas bien definidas para la solución de un problema. Los distintos algoritmos difieren entre sí por el medio en el cual el ajuste de los pesos sinápticos de las neuronas es formulado.

Paradigma de aprendizaje: tiene en cuenta el *modelo* de entorno en el cual la red se encuentra.

3.2. Algoritmos de Aprendizaje

3.2.1. Por Corrección de Error

Su principio básico es el de usar la *señal de error* ($y_{\text{deseada}} - y$) para modificar los pesos para gradualmente ir reduciendo el error. El aprendizaje ocurre entonces solo cuando se comete un error.

La *señal de salida* de una neurona k , denotada por $y_k(n)$, es comparada con la *respuesta deseada* $d_k(n)$, produciendo la *señal de error* $e_k(n)$:

$$e_k(n) = d_k(n) - y_k(n),$$

Esta *señal de error* funciona como un *mecanismo de control*, cuyo propósito es secuencialmente ajustar de forma correctiva los pesos sinápticos de la neurona k . De esa manera, $y_k(n) \rightarrow d_k(n)$. Esto se logra minimizando **función de costo** $\mathcal{E}(n)$ definida a partir de la señal de error. El proceso de aprendizaje termina cuando se alcanza un *estado estable*, donde los cambios en los pesos sinápticos son mínimos.

La minimización de $\mathcal{E}(n)$ lleva a la regla de aprendizaje conocida como **Regla Delta**, en la cual el ajuste $\Delta w_{kj}(n)$ aplicado al peso sináptico w_{kj} en la iteración n está definida por:

$$\Delta w_{kj}(n) = \eta e_k(n) x_j(n),$$

dónde η es una constante positiva que determina la *tasa de aprendizaje* al proceder de una iteración a la siguiente en el algoritmo de aprendizaje.

Esto presume que la señal de error es *conocida*, por lo que se debe proveer una respuesta deseada a la neurona k . Además, este tipo de aprendizaje es *local* a la neurona k .

La elección de η supondrá la futura estabilidad o convergencia del proceso iterativo de aprendizaje.

3.2.2. Basado en Memoria

En este aprendizaje la mayoría de las experiencias pasadas son guardadas explícitamente en la memoria como un par $\{\mathbf{x}_i, d_i\}_{i=1}^N$ {**patrón, respuesta deseada**}. Cuando aparece un nuevo patrón a clasificar \mathbf{x}_{test} , el algoritmo responde recuperando y analizando los datos que tiene almacenados en el *vecindario local* de \mathbf{x}_{test} .

Es necesario definir un **criterio** para establecer el vecindario local de \mathbf{x}_{test} (por ejemplo, *distancia euclídea*), y la **regla de aprendizaje** de los patrones de entrenamiento.

Ejemplos de clasificadores que usan este tipo de aprendizaje son:

- **Vecino más cercano** (*nearest neighbor*). La vecindad local está definida como el patrón \mathbf{x}_i más cercano a \mathbf{x}_{test} . Se le asigna entonces a este último la clase del correspondiente \mathbf{x}_i .
- **k -vecinos más cercanos** (*k-nearest neighbor*). La vecindad local está definida por los k patrones \mathbf{x}_i más cercanos a \mathbf{x}_{test} . Se le asigna a este último entonces la clase que aparece más frecuentemente en la vecindad.

3.2.3. Boltzmann

Un subconjunto de neuronas llamadas **visibles** interactúan con el entorno; el resto, llamadas **ocultas** (*hidden*), no. El objetivo es ajustar los pesos sinápticos para que el estado de las neuronas visibles satisfagan una distribución de probabilidad deseada.

3.2.4. Hebbiano

Aprendizaje bio-inspirado en el postulado de *Hebb*, según el cual, si dos neuronas interconectadas mediante una misma sinapsis son activadas simultáneamente, el peso de la sinapsis es incrementado. Por otro lado si estas neuronas se activan en tiempos diferentes, la sinapsis se debilita o se elimina.

Matemáticamente, considerando y_k y x_j como señales *post-sinápticas* y *pre-sinápticas* respectivamente, tenemos:

$$\Delta w_{kj}(n) = F(y_k(n), x_j(n)),$$

dónde F es una función que puede tomar muchas formas. Por ejemplo:

$$\Delta w_{kj}(n) = \eta y_k(n) x_j(n).$$

El aprendizaje es hecho de forma **local**: el cambio en los pesos sinápticos depende sólo de las actividades de las dos neuronas conectadas por él.

3.2.5. Competitivo

En este aprendizaje las neuronas de la RN compiten entre ellas para activarse. Mientras que en

una RN que se base en *aprendizaje Hebbiano* se pueden activarse más de una neurona al mismo tiempo, en el *aprendizaje competitivo* sólo se activa una a la vez.

Existen aquí tres elementos básicos:

- Un conjunto de neuronas iguales, con pesos sinápticos aleatoriamente distribuidos.
- Un *límite* impuesto en la “*fuera*” de cada neurona.
- Un mecanismo que permita hacer competir las neuronas en un modelo *winner-takes-all*.

Una neurona aprende moviendo sus pesos sinápticos en dirección al patrón de entrada. Matemáticamente:

$$\Delta w_{kj} = \begin{cases} \eta(x_j - w_{kj}), & \text{si la neurona } k \text{ gana la competencia,} \\ 0, & \text{si la neurona } k \text{ pierde la competencia.} \end{cases}$$

De esa manera, las distintas neuronas se especializan y se convierten en *clasificadores de características*.

3.3. Paradigmas de Aprendizaje

Hay cuatro paradigmas de aprendizajes principales:

Supervisado (*supervised*): la red es provista de una respuesta correcta (*output*) para cada entrada. Los pesos son determinados en base a permitir a la red producir salidas lo más cercanas a las respuestas correctas.

Aprendizaje reforzado (*reinforcement learning*): es una variante del aprendizaje *supervisado*, en el cual la red es provista de sólo una crítica acerca de lo correcto o no que es la salida de la red, no la respuesta correcta de por sí. El objetivo del aprendizaje es minimizar una *función de costo*.

Sin supervisar (*unsupervised*): no requiere una respuesta correcta asociada con cada entrada en el conjunto de entrenamiento. Explora la estructura interna de la información, y organiza los patrones en categorías en base a las correlaciones entre ellos.

Híbrido (*hybrid*): una parte de los pesos son determinados a través de un entrenamiento *supervisado*, y el resto son obtenidos a través de un aprendizaje *sin supervisar*.

Nota: Se puede considerar el aprendizaje *supervisado* dentro de la categoría de aprendizaje **con profesor**, mientras que el *reforzado* y *sin supervisar* dentro de aprendizaje **sin profesor**.

3.4. Tareas de aprendizaje

La elección de un **algoritmo de aprendizaje** depende fuertemente de la *tarea de aprendizaje* que la RNA debe llevar a cabo.

- **Asociación de patrones.** La memoria *asociativa* es una memoria distribuida bio-inspirada que aprende por *asociación*. Se utiliza *aprendizaje no-supervisado* si la relación es de **auto-asociación** (asociación de pares de patrones similares) o *aprendizaje supervisado* si es una relación de **hetero-asociación** (pares de patrones distintos).

- **Reconocimiento de patrones.** Los humanos son buenos en esta tarea, y lo logran a partir de un proceso de aprendizaje. La idea es clasificar patrones de entrada en categorías. Puede utilizar una **RNA sin supervisar** para *extracción de características* y luego una **RN supervisada** para *clasificación*; o una **RNA multicapa supervisada**.
- **Aproximación de funciones.** Con *aprendizaje supervisado*.
- **Control.** Mantener un proceso o parte crítica de un sistema bajo una condición controlada.
- **Filtrado.** Extraer información a partir de un conjunto de patrones de interés degradado por ruido.
- **Beamforming.** Es una especie de *filtrado espacial* que filtra a partir de comparaciones entre propiedades espaciales del objetivo y del ruido.

3.4.1. Aplicaciones de las RNAs

Las características de las **RNA** las hacen bastante apropiadas para aplicaciones en las que no se dispone *a priori* de un modelo identificable que pueda ser programado, pero se dispone de un conjunto básico de ejemplos de entrada (previamente clasificados o no). Asimismo, son altamente robustas tanto al ruido como a la disfunción de elementos concretos y son fácilmente paralelizables.

También se pueden utilizar cuando no existen modelos matemáticos precisos o algoritmos con complejidad razonable.

4. Perceptrón Simple

Es la forma más simple de RNA usada para clasificar patrones que sean **linealmente separables**, es decir, que estén ubicados en lados opuestos de un *híper-plano*. Básicamente, consiste en una sola neurona con pesos sinápticos ajustables y un *bias*.

Teorema de Convergencia del Perceptrón: Si los patrones usados para entrenar el perceptrón pertenecen a dos clases linealmente separables, entonces el algoritmo del perceptrón converge y posiciona una *superficie de decisión* en forma de un *híper-plano* entre las dos clases.

Si se requiere clasificar más clases linealmente separables, se pueden incluir más neuronas.

4.1. Algoritmo de aprendizaje

1. Se inicializan los pesos \mathbf{w} aleatoriamente dentro de un rango pequeño (por ejemplo, $[-0.5, 0.5]$).
2. Se muestran muchos patrones con las salidas esperadas y en cada caso se calcula:

$$y(n) = \phi(\langle \mathbf{w}(n), \mathbf{x}(n) \rangle)$$

\Rightarrow **SI** la salida de la red es **correcta** $\{y(n) = d(n)\}$, no se hacen cambios: *principio de mínima perturbación*.

\Rightarrow **SI** la salida de la red es **incorrecta** $\{y(n) \neq d(n)\}$, se hace una penalización: se actualiza \mathbf{w} en la dirección opuesta a la cual contribuyó a la salida incorrecta.

3. Volver a (2) hasta satisfacer algún criterio de finalización.

4.2. Entrenamiento por método de gradiente

La idea es a partir de una *función costo* $\mathcal{E}(\mathbf{w})$ que es una función *continuamente diferenciable*, encontrar el vector peso \mathbf{w}^* donde la función tenga su mínimo global. Una condición necesaria para esto es que: $\nabla \mathcal{E}(\mathbf{w}^*) = \mathbf{0}$, donde ∇ es el *operador gradiente*.

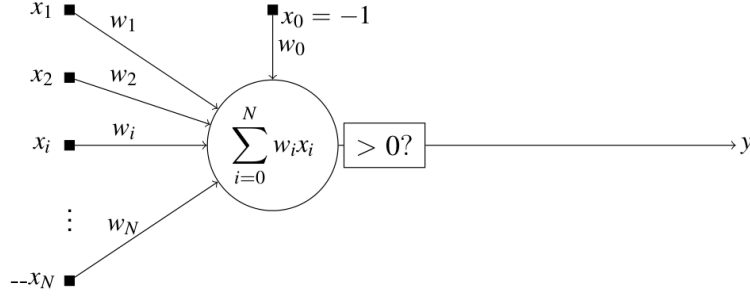


Figura 4: Modelo de una neurona. El bloque $\{> 0?\}$ es la función de activación ϕ .

En otras palabras, se requiere mover iterativamente los pesos \mathbf{w} en la dirección en que se reduce el error, la cual es opuesta a la dirección de su gradiente con respecto a los pesos. Matemáticamente:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu \nabla \mathcal{E}(\mathbf{w}(n)).$$

Nota: El perceptrón simple utiliza para esto el *algoritmo de LMS*, mientras que el multicapa utiliza el de *retropropagación*.

4.2.1. Algoritmo de Least-Mean-Square

Está basado en utilizar el *error instantáneo* para la *función de costo*, es decir:

$$\mathcal{E}(\mathbf{w}(n)) = \frac{1}{2} e^2(n),$$

dónde la señal de error en el tiempo n está definida como:

$$\begin{aligned} e^2(n) &= [d(n) - y(n)]^2 \\ &= [d(n) - \langle \mathbf{w}(n), \mathbf{x}(n) \rangle]^2. \end{aligned}$$

Entonces, aplicando el operador gradiente, obtenemos:

$$\begin{aligned} \nabla_{\mathbf{w}} e^2(n) &= 2 [d(n) - \langle \mathbf{w}(n), \mathbf{x}(n) \rangle] (-\mathbf{x}(n)) \\ &= 2 e(n) (-\mathbf{x}(n)). \end{aligned}$$

Reemplazando, llegamos a:

$$\nabla \mathcal{E}(\mathbf{w}(n)) = -e(n) \mathbf{x}(n).$$

Por lo que concluimos con qué:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n) \mathbf{x}(n).$$

4.2.2. Ejemplo gráfico de resolución del problema OR

En la **Figura 5**, tenemos una representación gráfica del problema de OR, a partir de la cual se pueden deducir los pesos para resolverlo al plantear el sistema de ecuaciones. En este caso, los resultados son $w_0 = -1$, $w_1 = w_2 = 1$. Como $y = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle)$, entonces $y = \text{sign}(x_1 + x_2 + 1)$ para trabajar con las clases -1 y 1 .

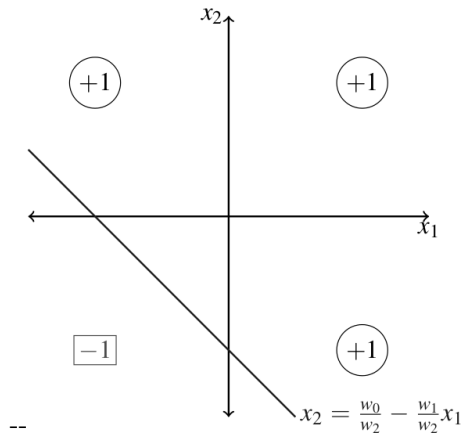


Figura 5: Representación gráfica del problema de OR.

5. Bases estadísticas de reconocimiento de patrones

5.1. Definiciones

- **Patrón:** objeto de interés que es identificable del resto, posiblemente difuso. Ejemplo: *huella digital*.
- **Objetivo del reconocimiento de patrones (RdP):** crear sistemas informáticos que imiten la percepción

y razonamiento humano: es decir, la capacidad de **distinguir** y **aislar** patrones, **agruparlos**, y **categorizarlos**. Posibles aplicaciones: reconocimiento de imágenes, y de habla y lenguaje.

5.2. Aproximaciones al reconocimiento de patrones

Existen diferentes **aproximaciones** al RdP, según sea la tarea: la **estadística** y la **sintáctica**.

Tipo	Aproximación Estadística	Aproximación Sintáctica
<i>Basado en...</i>	Teoría Estadística de la Decisión.	Teoría de Lenguajes Formales.
<i>Patrones representados...</i>	Como <i>vectores numéricos</i> .	Como <i>cadena de símbolos</i> .
<i>Representación de clases...</i>	Mediante <i>patrones prototipo</i> .	Mediante <i>reglas sintácticas</i> para especificar patrones válidos.
<i>Tipos de clasificadores</i>	Gaussianos, basados en distancia.	Autómatas, gramáticas, modelo oculto de Markov.

5.2.1. Aproximación Estadística

Con esta aproximación un patrón no es más que un **punto** en el **espacio de representación de los patrones**, que es un espacio de dimensionalidad determinada por el número de parámetros considerados. Esta aproximación concluye que es razonable que los patrones pertenecientes a una misma clase estén cercanos en el espacio de representación mientras que aquellos que pertenezcan a clases diferentes deberían estar en diferentes regiones del espacio de representación.

El estudio del conjunto apropiado de parámetros, la variabilidad de los patrones de una clase, las medidas de similitud entre patrones, y entre patrones y clases, constituye el **reconocimiento estadístico de patrones**.

5.2.2. Aproximación Sintáctica

No considera el **contexto**, esto es, la relación entre diferentes patrones: en ocasiones, patrones complejos pueden descomponerse recursivamente en patrones más simples hasta llegar a componentes básicos (de forma similar a como un texto se descompone en párrafos, frases, palabras y finalmente en letras).

Esta aproximación resulta muy adecuada para reconocimiento de voz, por ejemplo. Con esta aproximación, el problema se reduce a responder si un determinado patrón pertenece al lenguaje generado por una gramática. Estas técnicas se conocen como **reconocimiento sintáctico de patrones**.

5.3. Paradigma de trabajo funcional:

1. *Adquisición y pre-procesamiento de datos*: se adquiere un vector numérico que representa al patrón natural. Pertenecer en forma de un punto al *espacio de representación de patrones*.
2. *Extracción de características*: se extrae información *relevante* para la clasificación y se forma otro nuevo vector de **menor** dimensión que pertenece al **espacio de características**. Se desea que las características extraídas posean cualidades como:
 - + **Unicidad**: representaciones diferentes para objetos diferentes.
 - + **Precisión**: representación única para cada objeto.
 - + **Capacidad de generalización**.
 - + **Inmunidad al ruido**.
3. *Clasificación*: decisión sobre la clase.

5.4. Clasificador estadístico

Es una máquina formada por c **funciones discriminantes** $g_i : E \rightarrow \mathbb{R}$, $1 \leq i \leq c$, tal que dado un patrón $\mathbf{x} \in E$, se le asigna una clase w_i si $g_i(\mathbf{x}) > g_j(\mathbf{x})$, $\forall j \neq i$ (ver **Figura 6**).

El clasificador divide entonces el espacio E en c **regiones de decisión** R_1, R_2, \dots, R_c que contienen a los patrones de la correspondiente clase w_i . Las hipersuperficies del espacio que separan las regiones de decisión contiguas se denominan **fronteras de decisión** (FdD).

En los **clasificadores estadísticos** básicos, las FdD son combinaciones lineales o cuadráticas de las componentes del vector de características. De esa forma, obtengo fronteras de decisión *hiperplanas* o *hipercuadráticas* respectivamente.

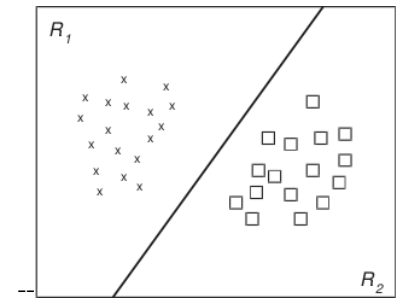
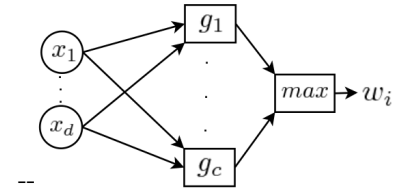


Figura 6: Arriba: Clasificador estadístico, representación gráfica. Abajo: ejemplo de frontera de decisión.

5.4.1. Tipos de probabilidades

Asumiendo que c es la etiqueta y \mathbf{x} es la muestra, tenemos que:

Tipo de probabilidad	Probabilidad de observar...
A priori	c sin conocer \mathbf{x} .
A posteriori	c conociendo \mathbf{x} .
Incondicional	\mathbf{x} sin conocer c .
Condicional	\mathbf{x} conociendo c .
Conjunta	\mathbf{x} con c .

5.4.2. Clasificador de Bayes

Regla de clasificación de Bayes: asignar a \mathbf{x} la clase con mayor *probabilidad a posteriori*:

$$\hat{w} = \max_{w_i: 1 \leq i \leq c} P(w_i | \mathbf{x}).$$

La probabilidad de que se incurra en un error puntual es: $P(\text{error}, \mathbf{x}) = 1 - \hat{w}$.

Clasificador de Bayes: utiliza las *probabilidades a posteriori* como *funciones discriminantes*: $g_i(\mathbf{x}) = P(w_i | \mathbf{x})$.

6. Perceptrón Multicapa

6.1. Introducción

La arquitectura del **Perceptrón Multicapa** (MLP) surge por corregir las limitaciones que las redes iniciales, *Adaline*⁵ y *Perceptrón* tenían, sobre todo en cuanto a separabilidad de funciones no-lineales.

Una de las ventajas del MLP es que es un **aproximador universal de funciones**, de modo que puede aproximar cualquier función continua en el espacio multidimensional real. Más concretamente, si posee al menos una *capa oculta* con suficientes neuronas, puede aprender cualquier tipo de función o relación continua entre un grupo de variables de entrada y salida.

⁵**Adaline** (*Adaptive Linear Neuron*) es una RNA similar al perceptrón. Es más preciso que este, y se diferencia del mismo en la etapa de *aprendizaje* por la forma de ajustar los pesos. La combinación de varios *adalines* es un **Madaline**.






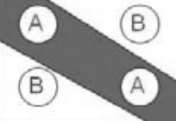


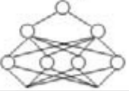



Estructura	Tipos de regiones de decisión	Problema XOR	Separación en clases	Formas regiones más generales
Una capa 	hemiplano limitado por hiperplano			
Dos capas 	Regiones convexas abiertas o cerradas			
Tres capas 	Arbitrarias (Complejidad limitada por N° de Nodos)			

Figura 7: Interpretación geométrica del rol de las neuronas ocultas en un espacio de dos entradas.

Por otra, el MLP es de relativo **fácil uso y aplicación**, dado que es una red sin recurrencias y *feed-forward*. Posee además una elevada capacidad de **generalización y robustez**.

La arquitectura del MLP está basado en una red *feed-forward* o con conexiones hacia delante, en la que se disponen de tres tipos de capas:

- La capa de **entrada**, en la que las neuronas actúan como búfer simplemente.
- Una o más capas **ocultas**.
- La capa de **salida**.

Todas las neuronas de la red (excepto las de la entrada, normalmente) llevan asociado un *umbral*.

Observación: Para la mayoría de los problemas, una sola *capa oculta* es suficiente. Dos *capas ocultas* son requeridas para modelar información con discontinuidades, pero puede introducir riesgos de convergencia en mínimos locales. No hay razones teóricas para usar más de dos *capas ocultas*.

El MLP posee tres características distintivas, cuya combinación con la capacidad de aprendizaje es lo que le da el poder computacional que lo caracteriza:

1. Todas las neuronas de la red poseen una *función de activación no-lineal* (por ejemplo, la *función sigmoidea*, que está biológicamente motivada).
2. Las *neuronas ocultas* le permiten a la red realizar tareas complejas al extraer progresivamente características de los patrones de entrada.
3. La red exhibe un alto grado de *conectividad*. En general, cada neurona de una capa tiene conexiones con todas las de la capa anterior.

6.2. Algoritmo de retropropagación

El entrenamiento del MLP se realiza utilizando el **algoritmo de retropropagación**.

6.2.1. Cálculo de las salidas en cada capa

Para reducir notación, considerar que estamos trabajando en la iteración n .

El campo local inducido $v_j(n)$ en la neurona j de la capa p es igual a:

$$v_j^{(p)} = \sum_{i=0}^m w_{ji}^{(p)} y_i^{(p-1)}$$

dónde m es la cantidad total de entradas más el

bias (los cuales conforman el vector \mathbf{x}), y $\mathbf{y}^{(0)} = \mathbf{x}$. De aquí, el valor $y_j^{(p)}$ que aparece a la salida de la neurona j de la capa p es igual a:

$$y_j^{(p)} = \phi(v_j^{(p)}).$$

Llamaremos al último $y_j^{(\text{output_layer})}$ como y_j (sin notación con superíndice).

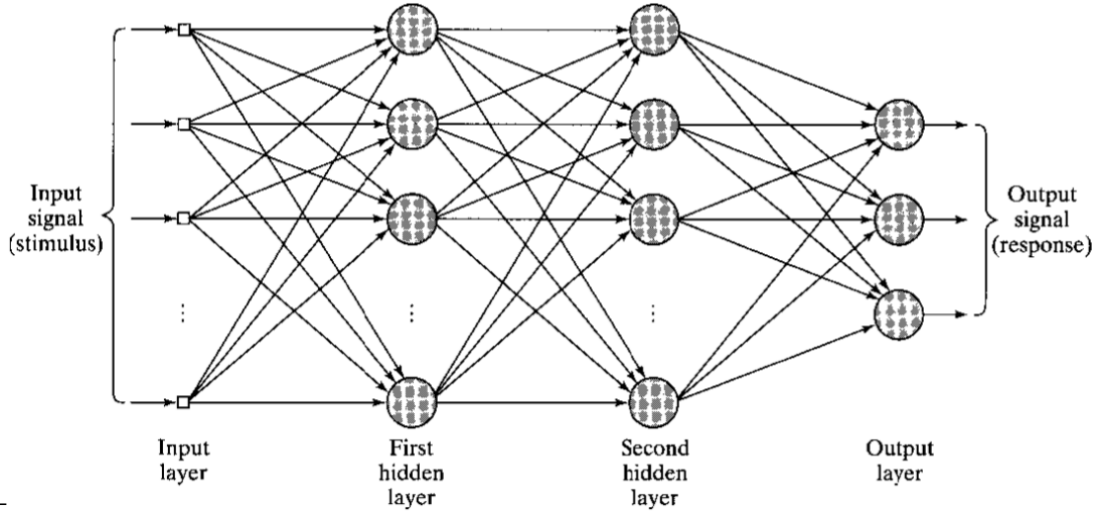


Figura 8: Arquitectura de un MLP con dos capas ocultas.

6.2.2. Criterio de Error

La señal de error a la salida de la neurona j que está en la capa de salida, en la iteración n , está dada por:

$$e_j(n) = d_j(n) - y_j(n).$$

A partir de sumar los valores de errores instantáneos de todas las neuronas de la capa de salida de la red (las únicas “visibles”, por lo que se puede conocer su error directamente), obtenemos la **suma del error cuadrático**:

$$\mathcal{E}(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n),$$

dónde C incluye a todas las neuronas de la capa de salida.

De manera similar al algoritmo LMS, se aplica una corrección $\Delta w_{ji}(n)$ a los pesos sinápticos w_{ji} , el cual es proporcional a la derivada parcial $\partial \mathcal{E}(n) / \partial w_{ji}(n)$, representativo de un *factor de sensibilidad* que indica la dirección de búsqueda en el espacio de pesos por el peso sináptico w_{ji} . Por *regla de la cadena*:

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \cdot \frac{\partial e_j(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)}.$$

Para el último término, tenemos qué:

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = \frac{\partial}{\partial w_{ji}(n)} \left\{ \sum_{i=0}^m w_{ji}(n) y_i(n) \right\} = \boxed{y_i(n)}$$

La corrección $\Delta w_{ji}(n)$ aplicada al peso $w_{ji}(n)$ es definida por la **regla delta** como:

$$\Delta w_{ji} = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \eta \left\{ -\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \cdot \frac{\partial e_j(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \right\} y_i(n) = \eta \delta_j(n) y_i(n)$$

dónde η es el parámetro de *tasa de aprendizaje* del algoritmo de retropropagación.

Si la *función de activación*⁶ está definida por $y_j(n) = \phi(v_j(n)) = 2 / \{1 + e^{-v_j(n)}\} - 1$, entonces:

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \frac{\partial}{\partial v_j(n)} \{ \phi(v_j(n)) \} = \frac{2}{(1 + e^{-v_j(n)})^2} \cdot e^{-v_j(n)} = \frac{1}{2} (y_j(n) + 1)(y_j(n) - 1).$$

A esta última expresión se llega mediante operaciones algebraicas y reemplazando por $y_j(n)$ cuando corresponda.

⁶Nota: la *función de activación* utilizada es *anti-simétrica* y comprende el rango $[-1, 1]$. Una función es *anti-simétrica* si $\phi(-v) = -\phi(v)$.

Desde aquí, consideramos dos casos diferentes, que tienen que ver con el hecho de que tanta información tenemos sobre la señal de error $e_j(n)$.

Caso 1: La neurona j está en la *capa de salida*

Puede aprovecharse el hecho de que se conoce la respuesta deseada para esa neurona j . Determinamos los valores de cada uno de los términos sin determinar de $\delta_j(n)$ diferenciando y sustituyendo apropiadamente:

$$\begin{aligned}\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} &= \frac{\partial}{\partial e_j(n)} \left\{ 1/2 \cdot \sum_{j \in C} e_j^2(n) \right\} = \boxed{e_j(n)} \\ \frac{\partial e_j(n)}{\partial y_j(n)} &= \frac{\partial}{\partial y_j(n)} \{d_j(n) - y_j(n)\} = \boxed{-1}\end{aligned}$$

Obtenemos la corrección del peso Δw_{ji} en la neurona j en la *capa de salida* como:

$$\begin{aligned}\Delta w_{ji}(n) &= \eta \delta_j(n) y_i(n) \\ &= \eta \left\{ -e_j(n) \cdot (-1) \cdot \left[\frac{1}{2} (y_j(n) + 1)(y_j(n) - 1) \right] \right\} y_i(n) \\ &= \boxed{\eta e_j(n) (y_j(n) + 1)(y_j(n) - 1) y_i(n)}\end{aligned}$$

Nótese que el término $y_i(n)$ hace referencia a la salida de la neurona i en la *capa oculta* anterior. Además, el término desapareció $1/2$ al incluirlo dentro del valor de η .

Caso 2: La neurona j está en la *capa oculta*

Aquí no hay un valor de salida deseado específico para la neurona. La señal de error ha de tener que ser entonces calculado recursivamente a partir de las señales de error de todas las neuronas conectadas a ella.

Cálculamos entonces el término que nos falta para calcular δ_j :

$$\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \cdot \frac{\partial e_j(n)}{\partial y_j(n)} = \frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

Trabajando los términos por separado, llegamos a que:

$$\begin{aligned}\frac{\partial e_k(n)}{\partial v_k(n)} &= \frac{\partial}{\partial v_k(n)} \{d_k(n) - \phi_k(v_k(n))\} = \boxed{-\phi'_k(v_k(n))} = \frac{1}{2}(1 + y_k(n))(1 - y_k(n)) \\ \frac{\partial v_k(n)}{\partial y_j(n)} &= \frac{\partial}{\partial y_j(n)} \left\{ \sum_{j=0}^m w_{kj}(n) y_j(n) \right\} = \boxed{w_{kj}(n)}\end{aligned}$$

Volviendo a juntar, y conociendo el valor de la derivada de la función de activación, tenemos:

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} = \frac{1}{2} \sum_k e_k (y_k(n) + 1)(y_k(n) - 1) w_{kj}(n)$$

Entonces, para simplificar, calculamos δ_j en la *capa oculta* p durante la iteración n como:

$$\begin{aligned}\delta_j^{(p)}(n) &= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= -\left[\frac{1}{2} \sum_k e_k (1 + y_k(n))(1 - y_k(n)) w_{kj}(n) \right]^{(p+1)} \cdot \left[\frac{1}{2} (y_j(n) + 1)(y_j(n) - 1) \right]^{(p)} \\ &= \left[\sum_k \delta_k(n) w_{kj}(n) \right]^{(p+1)} \cdot \left[\frac{1}{2} (1 + y_j(n))(1 - y_j(n)) \right]^{(p)}\end{aligned}$$

Nota: Los superíndices (p) , $(p+1)$ indican de las neuronas de **qué** capa salen los valores de los términos a utilizar en cada momento.

Finalmente, calculamos Δw_{ji} de la neurona j en la capa p durante la iteración n como:

$$\begin{aligned}\Delta w_{ji}^{(p)}(n) &= \eta \delta_j^{(p)}(n) y_i^{(p-1)}(n) \\ &= \eta \left[\sum_k \delta_k(n) w_{kj}(n) \right]^{(p+1)} \cdot [(1 + y_j(n))(1 - y_j(n))]^{(p)} y_i^{(p-1)}(n)\end{aligned}$$

De la misma forma que antes, el término $1/2$ fue absorbido por la *tasa de aprendizaje* η .

6.2.3. Resumen del algoritmo de retropropagación

1. Inicialización aleatoria de los pesos.
2. Propagación hacia delante (de la entrada), calculando el valor de los sucesivos y_j de cada neurona j de forma iterativa.
3. Propagación hacia atrás (del error). Aquí se calcula el valor de δ_j correspondiente a cada neurona j , según si pertenece ésta a la *capa oculta* o a la *capa de salida*.
4. Adaptación de los pesos hacia delante, usando el valor de δ_j ya calculado.
5. Iteración: vuelve a (2) hasta convergencia o finalización.

6.3. Tasa de Aprendizaje: Constante de Momento

El algoritmo de retropropagación provee una “*aproximación*” a la trayectoria en el espacio de pesos computada por el *método del gradiente descendiente*. Si η es chico, más pequeños son los cambios en los pesos sinápticos, pero más lento es el aprendizaje. Si η es grande, resulta en grandes cambios en los pesos sinápticos, pero puede provocar *inestabilidad* (oscilaciones). Para solventar los problemas, agregamos una **constante de momento**:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) - \eta \frac{\partial \mathcal{E}(t)}{\partial w_{ji}(t)}.$$

La inclusión del término momento influye de la siguiente forma:

- Cuando $\partial \mathcal{E}(t)/\partial w_{ji}(t)$ tiene el mismo signo en sucesivas iteraciones: $\Delta w_{ji}(n)$ crece en magnitud y los $w_{ji}(n)$ se ajustan por un gran valor. El **momento** acelera la convergencia del método en direcciones largas de descenso.
- Si $\partial \mathcal{E}(t)/\partial w_{ji}(t)$ tiene diferente signo en sucesivas iteraciones: el $\Delta w_{ji}(n)$ disminuye en magnitud y los $w_{ji}(n)$ se ajustan por un pequeño valor. El **momento** tiene un efecto *estabilizante* en regiones oscilantes y ayuda a evitar a que se quede en un mínimo local.

6.4. Algunas heurísticas para mejorar la *performance*

1. Utilizar un modo **secuencial** (actualiza iterativamente por cada patrón) de retropropagación es más rápido que en **lote**.
2. Usar patrones de ejemplo diversos, que permitan buscar en un mayor espacio de pesos. También conviene aleatorizar el orden de las entradas, para evitar que sucesivas entradas pertenezcan a la misma clase.
3. Funciones de activación **anti-simétricas** proveen un aprendizaje más rápido que las **simétricas**.
4. Normalizar entradas, eliminando correlaciones entre ellas, y escalando las covarianzas para sean similares.

5. Inicializar pesos con valores ni muy pequeños (poca dispersión inicial en el espacio de búsqueda), ni muy grandes (provocan saturación).
6. La tasa de aprendizaje debe ser *idealmente* tal que todas las neuronas aprendan a la misma velocidad.

6.5. Generalización

Una red se dice que **generaliza** cuando el mapeo entrada/salida computado por la red es aproximadamente correcto para **patrones de prueba** que la red nunca ha “visto” durante su entrenamiento. Sin embargo, si una RNA aprende demasiado los ejemplos, puede terminar memorizando el lote de entrenamiento; es decir, se produce un **sobre-entrenamiento**, en dónde se pierde la habilidad de generalización.

La generalización está influida por tres factores:

1. El tamaño del lote de entrenamiento;
2. la arquitectura de la RNA;
3. y la complejidad física del problema a resolver.

Dado que el último factor no se puede modificar, debemos manipular los dos primeros. En este contexto, si uno está fijo, es el otro el que hay que tratar de modificar para conseguir el mejor poder de generalización.

6.6. Validación Cruzada

Se utiliza en entornos donde el objetivo principal es tener una gran RNA con una muy buena capacidad de generalización, al elegir para ello el “mejor” conjunto de patrones para entrenar la red.

Para ello, se parten los datos disponibles entre un **conjunto de entrenamiento** y uno **de prueba**. A su vez, el primero se subdivide en:

- **Conjunto de estimación**, usado para seleccionar el modelo.
- **Conjunto de validación**, usado para validar el modelo.

La motivación aquí es la de validar el modelo en un conjunto de datos diferentes del usado para la estimación de parámetros (pesos). De esta forma, usamos el conjunto de entrenamiento para medir la *performance* de los distintos modelos candidatos, y elegir el “mejor”.

6.6.1. Método de Detención Temprana de Entrenamiento

Existe la posibilidad de que el modelo elegido *sobre-entrene* la red. Para evitarlo, se revisa cada cierta cantidad de épocas de entrenamiento el error que provoca probar la RNA con el *conjunto de validación*. Cuando la *performance* con este conjunto deja de mejorar, el algoritmo **se detiene**. Es entonces cuando se prueba la RNA con el *conjunto de prueba*, para medir la capacidad de generalización de la red.

Este procedimiento de detener el entrenamiento se conoce como método de la detención temprana *early-stopping*.

6.6.2. Características

- Es rápido y el usuario solo debe estimar la *proporción de casos de validación a usar* (tarea que no es trivial).

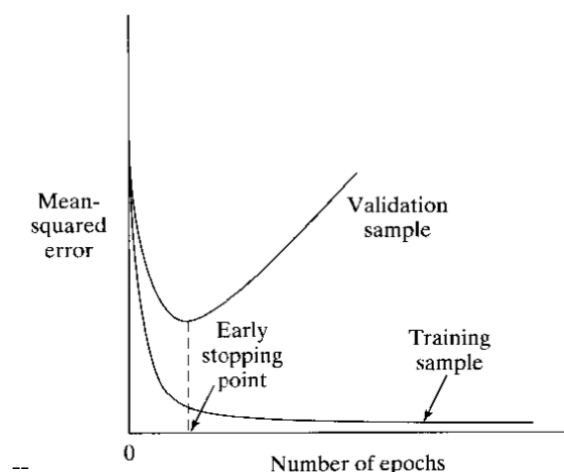


Figura 9: Ilustración del método de detención temprana. El *training sample* corresponde al conjunto de *estimación*.

- El resultado depende altamente de cuales patrones han sido agrupados en los conjuntos.
- Dado a que la *curva de error de validación* puede no ser siempre descendente, no está claro cuando detener el aprendizaje. Se puede entrenar hasta convergencia, y luego elegir la iteración con el menor error de validación, pero esto elimina la ventaja de la rapidez.

6.6.3. Variantes de la Validación Cruzada

La aproximación descripta es la *hold out method*. En ocasiones en que se tienen pocos ejemplos etiquetados, se puede utilizar una variante llamada *multifold cross-validation*.

Básicamente, divide el conjunto de N ejemplos disponibles en K subconjuntos del mismo tamaño. Entonces se utilizan $K - 1$ conjuntos para formar el *conjunto de estimación*, y el que queda el de *validación*. Esto se repite hasta que todos los subconjuntos fueron en algún momento el conjunto de validación. El *performance* del modelo se obtiene al promediar el error cuadrático en cada iteración de selección.

El método *leave-one-out* se usa en caso de extrema falta de ejemplos. Aquí, el tamaño del K utilizado es 1.

7. Red Neuronal de función de Base Radial

7.1. Introducción

Los métodos que utilizan las RN de *función de base radial* (RBF) tienen sus orígenes en técnicas para realizar la interpolación exacta de un conjunto de N puntos multidimensionales dados de una cierta función $\phi : \mathbb{R}^N \rightarrow \mathbb{R}$, tal que la función interpolante sea (μ_j es un vector):

$$h(\mathbf{x}) = \sum_{j=1}^N w_j \phi(\|\mathbf{x} - \mu_j\|)$$

dónde se utiliza normalmente como función de base radial una gaussiana multidimensional $\phi(k) = \exp(-k^2/2\sigma^2)$, siendo σ un parámetro que controla la **suavidad** de la función interpolante. La función gaussiana es una **función localizada**, es decir cumple con la propiedad de que $\phi(x) \rightarrow 0$ cuando $|x| \rightarrow \infty$.

Por otro lado, cuando utilizamos las funciones anteriores para armar una RN con RBF utilizamos el siguiente **modelo matemático**:

$$y_k(\mathbf{x}_l) = \sum_{j=1}^M w_{kj} \phi_j(\mathbf{x}_l), \quad \text{con } \phi_j(\mathbf{x}_l) = \exp\left(-\frac{\|\mathbf{x}_l - \mu_j\|^2}{2\sigma_j^2}\right)$$

dónde \mathbf{x} es el vector de entrada, μ es el centroide, y σ el factor suavizante o ancho.

7.1.1. Construcción

En su forma más básica, consiste en tres capas con diferentes roles:

- la **capa de entrada**, que posee los nodos de entrada que une la RNA con el entorno;
- la segunda capa, la única *oculta*, en la que se utiliza un *aprendizaje no supervisado* y se seleccionan las regiones interesantes;
- y en la tercera capa, en donde se usa un aprendizaje *supervisado*, que asigna alguna clase a esas regiones.

La *función de activación* en la segunda capa es una *gaussiana*: da una activación grande cuando el patrón \mathbf{x}_l se encuentra en cierto rango. La tercera capa toma todas las salidas de la segunda capa y las pondera.

Al estar cada capa conformada de neuronas de distintos tipos, se hace un **entrenamiento por separado**.

El **entrenamiento** se realiza en dos etapas: en la primera, se entrena la segunda capa de la red buscando los valores adecuados de σ y μ para cada neurona; en la segunda etapa, estos valores permanecen fijos y se entrenan los pesos de la tercera capa.

En la práctica, existe un intercambio entre usar un número de RBFs con mayor cantidad de parámetros, o de utilizar un número mayor de RBFs más sencillas.

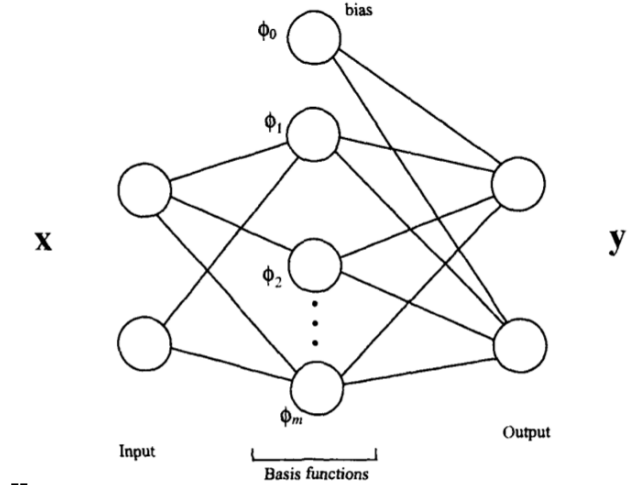


Figura 10: Arquitectura de una red neuronal RBF.

7.2. Entrenamiento en la Capa RBF

Suele utilizarse un algoritmo de **clustering no supervisado** conocido como **k-means**. Su objetivo es encontrar k conjuntos disjuntos C_j tales que los patrones de cada conjunto sean lo más parecidos entre sí, y que cada conjunto sea lo más diferente de los demás. Existen dos versiones de este algoritmo: *batch* y *online*. El número k ha de ser establecido de antemano por el usuario.

Una vez que los diferentes centroides de cada conjunto han sido establecidos, la varianza de cada función puede calcularse como la varianza de los puntos de cada conjunto.

7.2.1. Algoritmos de adaptación

Batch k-means:

1. *Inicialización*: se forman los k conjuntos $C_j(0)$ con patrones \mathbf{x}_l elegidos arbitrariamente.
2. Se calculan los centroides:

$$\mu_j(n) = \frac{1}{|C_j(n)|} \sum_{l \in C_j(n)} \mathbf{x}_l$$

3. Se reasignan los \mathbf{x}_l al C_j más cercano:
$$l \in C_j \iff \|\mathbf{x}_l - \mu_j\|^2 < \|\mathbf{x}_l - \mu_i\|^2, \forall i \neq j.$$
4. Volver a (2) hasta que no haya reasignaciones.

Online k-means:

1. *Inicialización*: se eligen k patrones aleatoriamente y se usan como centroides iniciales $\mu_j(0) = \mathbf{x}_l'$.
2. Selección de un nuevo patrón \mathbf{x}_l :

$$j^* = \arg \min_j \{\|\mathbf{x}_l - \mu_j\|\}.$$

3. Adaptación según cual μ_j esté más cerca:

$$\mu_{j^*}(n+1) = \mu_{j^*}(n) + \eta(\mathbf{x}_l - \mu_{j^*}(n)).$$

4. Volver a (2) hasta no encontrar mejoras significativas en j .

7.3. Entrenamiento en la tercera capa

7.3.1. Entrenamiento de pesos usando LMS

Se procede de igual manera que en un perceptrón simple, sabiendo que $e_k(n) = y_k(n) - d_k(n)$:

$$\begin{aligned}\mathcal{E}(n) &= \frac{1}{2} \sum_k e_k^2(n) = \frac{1}{2} \sum_k \left(\sum_j w_{kj}(n) \phi_j(n) - d_k(n) \right)^2, \\ \frac{\partial \mathcal{E}(n)}{\partial w_{kj}(n)} &= (y_k(n) - d_k(n)) \cdot \frac{\partial}{\partial w_{kj}} \left(\sum_j w_{kj} \phi_j(n) - d_k(n) \right), \\ \frac{\partial \mathcal{E}(n)}{\partial w_{kj}(n)} &= e_k(n) \phi_j(n)\end{aligned}$$

Quedando de esta manera la regla de aprendizaje:

$$w_{kj}(n+1) = w_{kj}(n) - \eta \cdot \frac{\partial \mathcal{E}(n)}{\partial w_{kj}(n)} = w_{kj}(n) - \eta e_k(n) \phi_j(n).$$

7.3.2. Entrenamiento de pesos usando la pseudo-inversa del vector $\phi(\mathbf{x}_l)$

Se puede optimizar los pesos minimizando una **función de error** apropiada. Por ejemplo, se puede usar la siguiente función de error:

$$E = \frac{1}{2} \sum_n \sum_k \{y_k(\mathbf{x}^n) - t_k^n\}^2$$

Como es una función cuadrática de los pesos, su mínimo puede ser encontrado en términos de la solución de un conjunto de ecuaciones lineales: $G^S G W^S = G^S S$.

La solución viene dada por la expresión: $W = G^+ S$ (con $+$ representando la pseudo-inversa, dada por $G^+ = (G^T G)^{-1} G^T$), donde W es la matriz de orden $(n+1) \cdot r$ que posee los n pesos y los umbrales en la última fila. La matriz G posee todas las funciones de activación para cada uno de los patrones de entrada, es de orden $(n+1) \cdot N$, siendo $g_{in} = \phi_i(n)$ y ϕ_i la función de activación de la neurona oculta i para el patrón de entrada $X(n)$. S es la matriz de salidas deseadas de la red, de orden $N \cdot r$.

Un problema que puede surgir es que la matriz pseudo-inversa esté mal condicionada, por lo que se utiliza una descomposición en valores singulares para resolverla.

7.4. Comparación entre las RBF y los MLP

Para más diferencias, ver **Tabla 1**.

- | | |
|--|---|
| <ul style="list-style-type: none">■ Ambos son redes no-lineales de propagación hacia delante en capas.■ Ambos son aproximadores universales, por lo que ambos se pueden utilizar para resolver los mismos tipos de problemas.■ Las neuronas en las capas ocultas sirven a propósitos distintos en cada tipo de red.■ RBF usa la norma euclídea como argu- | <p>mento de la función de activación, mientras que el MLP usa el producto punto.</p> <ul style="list-style-type: none">■ El RBF converge <i>linealmente</i>, tiene un entrenamiento más rápido, arquitectura más simple, y una combinación de diferentes paradigmas de aprendizaje.■ El MLP determina todos sus parámetros luego de un único entrenamiento global. El RBF requiere dos etapas de entrenamiento. |
|--|---|

8. Redes Competitivas

Las **redes competitivas** son muy utilizadas en la resolución de problemas de clasificación, aunque en algunos casos también se utilizan para la extracción de características relevantes, sobre todo aplicados a aprendizajes *no supervisados*.

En general, no suelen tener arquitecturas muy complejas, y en muchas ocasiones constan únicamente de dos capas: la **capa de entrada** y la llamada **capa de competición**, que todas ellas disponen. En esta última capa, se utiliza un **aprendizaje competitivo**, en el que las neuronas compiten entre ellas para ser activadas, con el resultado de que una única neurona o un conjunto de neuronas es activada por vez.

8.1. Mapas Auto-Organizativos

Los **mapas auto-organizativos** (SOMs) presentan la propiedad bio-inspirada de *preservación topológica*: el cerebro humano está organizado en varios lugares de modo tal que diferentes entradas sensoriales (*táctil, visual, acústica*) son mapeadas en diferentes áreas de la corteza cerebral ordenadas topológicamente.

Los SOM, como su nombre indica, se organizan solos, es decir, como parte del aprendizaje, al activarse una neurona correspondiente a un ejemplo, la neurona evoluciona en el espacio, moviéndose hacia el punto del hiperespacio de entrada en el que está el ejemplo.

Este movimiento, sin embargo, no sólo afecta a la neurona activa, sino a las de su vecindario, es decir, las neuronas de alrededor, que son desplazadas ligeramente con esta, como si de una malla se tratara. De este modo, todas las neuronas se irán desplazando por el espacio de entrada.

Es entonces el SOM caracterizado por la formación de un *mapa topográfico*, donde se tiene el:

«*Principio de formación de mapas topográficos*: La ubicación espacial de una neurona en un mapa topográfico corresponde a un dominio o característica particular de los datos extraídos del espacio de patrones de entrada.»

Aplicaciones: Los SOM nos permiten detectar relaciones entre los ejemplos y características comunes. También se utilizan para compresión y optimización de datos.

8.1.1. Dos modelos básicos de mapeo

Existen dos modelos de mapas auto organizativos, el de **Willshaw-von der Malsburg** y el de **Kohonen**. El primero está inspirado en el funcionamiento de la retina junto a la corteza visual en los vertebrados; mientras que el segundo puede ser o derivado a partir de una motivación neurobiológica (acercamiento tradicional) o, alternativamente, a partir de una aproximación basada en un esquema de codificador-decodificador. Este último modelo, el de Kohonen, es el más difundido.

8.1.2. Algoritmo de formación

El **principal objetivo** de un SOM es transformar los patrones de entrada de dimensión arbitraria en un mapa discreto uni- o bi-dimensional, y de hacer esta transformación adaptativamente de una forma topológicamente ordenada.

El algoritmo responsable de la formación del SOM procede primero *inicializando* los pesos sinápticos en la red con valores aleatorios pequeños. Una vez que se ha hecho esto, existen tres procesos esenciales involucrados en la formación SOM: **competición, cooperación y adaptación**.

Proceso de Competición:

Sea m la dimensión de los patrones de entrada y l la cantidad de neuronas de la grilla, entonces, se representan correspondientemente:

$$\begin{aligned} \text{el patrón de entrada:} & \quad \mathbf{x} = [x_1, x_2, \dots, x_m]^T, \\ \text{el peso sináptico de la neurona } j: & \quad \mathbf{w}_j = [w_{j1}, w_{j2}, \dots, w_{jm}]^T, \quad j = 1, 2, \dots, l. \end{aligned}$$

La neurona ganadora $i(\mathbf{x})$ será aquella que minimice la distancia al patrón de entrada \mathbf{x} :

$$i(\mathbf{x}) = \arg \min_j \|\mathbf{x} - \mathbf{w}_j\|, \quad j = 1, 2, \dots, l.$$

Proceso Cooperativo:

La neurona ganadora se localiza en el centro de un vecindario topológico de neuronas cooperativas. Este vecindario bio-inspirado se define por la *interacción lateral* entre las neuronas excitadas.

Sea $h_{j,i}$ el vecindario centrado en la neurona ganadora i , que abarca un conjunto de neuronas vecinas j . Sea $d_{j,i}$ la distancia *lateral* entre la neurona ganadora i y la neurona excitada j . La función de vecindario $h_{j,i}$ es función de la distancia lateral y satisface los siguientes requerimientos:

- $h_{j,i}$ tiene su máximo valor en la neurona ganadora i en la cual $d_{j,i} = 0$;
- y la amplitud de $h_{j,i}$ decrece monotónicamente con la distancia lateral y $h_{j,i} \rightarrow 0$ cuando $d_{j,i} \rightarrow 0$. Esta condición es necesaria para la **convergencia**.

Una función que satisface estos requerimientos es, por ejemplo, la *función gaussiana*:

$$h_{j,i}(\mathbf{x})(n) = \exp\left(-\frac{d_{j,i}^2}{2\sigma^2(n)}\right),$$

la cuál es independiente de la localización de la neurona ganadora (*invariante a la traslación*). El parámetro σ controlará **cuánto** participarán las neuronas vecinas en el proceso de aprendizaje. Como el vecindario topológico $h_{j,i}$ debe **reducirse** con el tiempo (otra característica del SOM), la constante σ aparece en la expresión como una función **variable** en el tiempo ($\sigma(n)$). Para esto puede utilizar el **decaimiento exponencial**⁷.

Nota: La distancia lateral se mide en el espacio de salida, no en el espacio multidimensional de entrada.

Proceso Adaptativo:

La idea de este proceso es que los pesos sinápticos de las neuronas puedan cambiar en relación con el patrón de entrada dado. La idea entonces es mover el vector de pesos sinápticos de la neurona ganadora (y el de todas sus vecinas en el vecindario) en la dirección del patrón procesado:

$$\mathbf{w}_j(n+1) = \mathbf{w}_j(n) + \eta(n) h_{j,i}(\mathbf{x})(n)(\mathbf{x} - \mathbf{w}_j(n)).$$

La tasa de aprendizaje η debe decrecer con el tiempo (por ejemplo, usando *decaimiento exponencial*).

Dentro del proceso de adaptación tenemos **dos fases de entrenamiento** con objetivos distintos:

Ordenamiento. Ocurre el ordenamiento topológico de los pesos. Dura cerca de 1000 iteraciones, y a veces más.

permanecer por arriba de 0.01.

- La η debe comenzar con un valor cercano a 0.1; y luego decrecer gradualmente, pero

- La función vecindad $h_{j,i}(n)$ debería incluir inicialmente toda las neuronas de la red, centrada en la neurona ganadora i , y luego achicarse con el tiempo.

⁷El *decaimiento exponencial* de una variable α durante la iteración n puede verse como: $\alpha(n) = \alpha_0 \cdot \exp(-n/C)$, donde C es una constante de tiempo y α_0 el valor inicial de α .

Convergencia. Ocurre el ajuste fino para asegurar una precisa cuantificación estadística. La cantidad de iteraciones es cercana a 500 veces el número de neuronas en la red.

- La η debe mantenerse a un pequeño valor, del orden de 0.01. No debe reducirse nun-

ca a cero, porque puede llevar a un estado *metaestable*.

- La función vecindad $h_{j,i}(n)$ debería incluir sólo las neuronas más cercanas, e incluso solamente a la neurona ganadora i .

Nota: a veces se incluye además una fase de *transición* con valores intermedios.

Tipo	MLP	RBFNN	SOM
<i>Regla de Aprendizaje</i>	Corrección de Error.	Corrección de Error/Competitivo.	Competitivo.
<i>Arquitectura</i>	Tiene p capas ocultas.	Una capa oculta.	Neuronas dispuestas en una malla.
<i>Función de activación</i>	No lineales.	No lineal y lineal (<i>gaussiana</i>).	No lineales y distancia dependientes.
<i>Salida depende de la activación...</i>	De todas las neuronas.	De las neuronas cercanas al patrón de entrada.	De las neuronas cercanas al patrón de entrada en el mapa topológico.
<i>Clasificación de regiones</i>	Separadas por curvas (<i>sigmoidea</i>) ó polígonos (<i>signo</i>).	Por clústeres.	No clasifica regiones, las ordena por ubicación espacial.

Cuadro 1: Tabla comparativa entre perceptrones multicapa, redes de funciones de base radial, y mapas auto-organizativos.

8.2. Learning Vector Quantification (LVQ)

La **cuantización vectorial** una técnica que aprovecha la estructura intrínseca de los vectores de entrada con propósitos de **compresión de datos**. El espacio de entrada se divide en distintas regiones y por cada región se define un **vector de reconstrucción**. Cuando se presenta un vector al cuantizador, este analiza la región a la que pertenece y representa el vector de entrada con el *vector de reconstrucción* de dicha región. De esta manera, se comprimen los vectores de múltiples dimensiones a números enteros que indican a la “clase” (vector de reconstrucción) que pertenecen. El conjunto de todos los vectores de representación se denomina **code-book** y cada uno de sus elementos se denomina **code-block**.

Este cuantizador puede entrenarse de diferentes maneras: usando SOM, *k-means*, o también el algoritmo **LVQ**. Este último es una técnica de *aprendizaje supervisado* que usa información de la clase **real** del patrón de entrada para mover el **vector de reconstrucción** lentamente para mejorar la calidad de las regiones de clasificación.

Se elige un vector de entrada \mathbf{x} , si la clase del vector y el *vector de reconstrucción* coinciden, entonces se mueve este último en la dirección del patrón, caso contrario, se lo mueve en la dirección opuesta.

8.2.1. Algoritmo de aprendizaje básico

- Tomamos \mathbf{m}_i vectores de reconstrucción. Puede haber uno o más vectores de reconstrucción por clase, usualmente se usa más de uno.
- Sea $c = \arg \min_i (||\mathbf{x} - \mathbf{m}_i||)$ la distancia definida entre el vector de entrada \mathbf{x} y el más cercano \mathbf{m}_i , denotado por \mathbf{m}_c . Cuando clasificamos, \mathbf{x} es usualmente tomado para que pertenezca a la clase más cercana a su vector prototipo, pero el \mathbf{m}_c también es usado en la fase de aprendizaje.

- Ahora el aprendizaje procede de la siguiente manera (modificando sólo el $\mathbf{m}_c(t)$ más cercano a la entrada $\mathbf{x}(t)$ en la iteración t):

$$\begin{aligned}\mathbf{m}_c(t+1) &= \mathbf{m}_c(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{m}_c(t)], & \text{si } \mathbf{x} \text{ y } \mathbf{m}_c \text{ pertenecen a la misma clase,} \\ \mathbf{m}_c(t+1) &= \mathbf{m}_c(t) - \alpha(t)[\mathbf{x}(t) - \mathbf{m}_c(t)], & \text{si } \mathbf{x} \text{ y } \mathbf{m}_c \text{ pertenecen a distintas clases.}\end{aligned}$$

- El $0 < \alpha(t) < 1$ es la *tasa de aprendizaje*. Puede ser una constante o decrecer con el tiempo. Por ejemplo, en el LVQ1 se toma un α inicialmente menor a 0.1 y decrece linealmente con el tiempo. El hecho de que sea < 1 hace que la importancia relativa de los primeros patrones sea **menor** que la de los últimos: es por ello que lo hacemos decrecer en el tiempo para que afecte por igual a todos los patrones.
- Para conseguir un α_c óptimo para cada centroide, se debe cumplir que:

$$\alpha_c(n) = [1 - s(n) \alpha_c(n)] \cdot \alpha_c(n-1)$$

dónde $s(n)$ vale 1 si \mathbf{x} y \mathbf{m}_c son de la misma clase, o -1 en caso contrario.

- La iteración se detiene generalmente cuando ha pasado un número máximo de iteraciones, cuando α alcanza un valor mínimo, ó cuando la tasa mínima de error de entrenamiento es alcanzada.

8.2.2. Diferencias con el SOM

El LVQNN es una red **supervisada**, **sin estructura topológica** (cada neurona de salida representa una categoría conocida), y **sin vecindario** (sólo se activa la neurona ganadora).

9. Redes Neuronales Dinámicas

9.1. Introducción

El *tiempo* constituye un ingrediente esencial en el proceso de aprendizaje. Puede ser *continuo* o *discreto* (el que usamos normalmente). Es a través de la incorporación del tiempo dentro de la operación de una red neuronal que se es posible seguir variaciones estadísticas en procesos **no estacionarios** tales como señales de voz, de radar, fluctuaciones de mercado, etc.

Para ello, tomamos una red “estática” (el MLP por ejemplo), y le imbuímos propiedades dinámicas: le otorgamos **memoria**. La misma puede ser dividida dependiendo el tiempo de retención, en:

- **Memoria de largo-plazo:** cuando trabajamos con aprendizaje *supervisado* y con todos los datos a disposición.
- **Memoria de corto-plazo:** cuando la tarea cambia en el tiempo.

Una simple manera de incorporar la *memoria a corto-plazo* es a través del uso de *time-delays* (los z^{-1}), que pueden ser implementados a nivel sináptico dentro de la red (o en la capa de entrada). El uso de *time-delays* está neuro-biológicamente motivado, ya que es bien sabido que existen desfases en el cerebro, y que estos juegan un papel fundamental en el procesamiento de la información.

9.2. Clasificación

Las redes neuronales dinámicas (RND) pueden clasificarse en:

Redes con Retardo en el Tiempo (TDNN). Siguen siendo redes *feed-forward*. Incorporan la cuestión temporal agregando retardos en las entradas. De esta manera la red adquiere una especie de memoria.

Redes Recurrentes (RNN). Aparece el *feed-back*. Basicamente, poseen dos utilidades: como *memorias asociativas* y como *mapeos de entrada/salida*. A la vez se subdividen en:

- **totalmente recurrentes**, donde no existe restricción de conectividad. Ej: Hopfield, Boltzman, ART;
- **y parcialmente recurrentes**, que poseen algunas conexiones recurrentes. Ej: BPTT, Elman, Jordan.

9.3. Red de Hopfield

La red de *Hopfield* se **engloba** dentro de las redes **totalmente recurrentes**. Consiste en un grupo de neuronas y su correspondiente conjunto de unidades de retraso, formando un *multiple-loop feed-back system*. Básicamente, la salida de cada neurona es retroalimentada, con un cierto desfase de una unidad, al resto de las neuronas de la red (no hay *self-feed-back*).

9.3.1. Características

El funcionamiento de la red es el de una **memoria asociativa**, permitiendo el almacenamiento y recuperación de patrones incluso cuando estos están incompletos. Los patrones que procesa son estáticos, aunque su procesamiento es temporal, es decir, la red requiere de un cierto tiempo de evolución hasta ofrecer la salida.

La filosofía de la red *Hopfield* es diferente al del resto de las redes. Su arquitectura carece de entradas o salidas en forma de neuronas, más bien se habla del **estado** de la red. Es decir, se inicializan las neuronas a un cierto valor y la red evoluciona hacia un estado estable, que corresponderá con la salida asociada a la entrada.

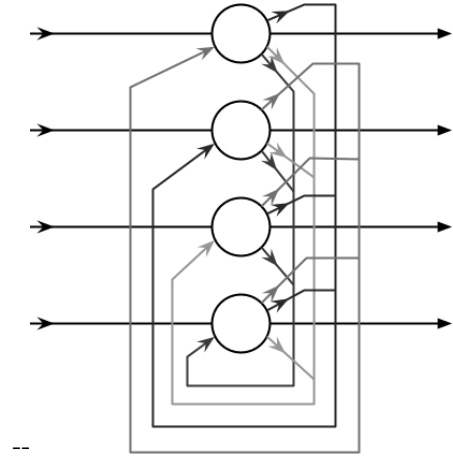


Figura 11: Arquitectura de una red de Hopfield.

9.3.2. Fases de Operación

1) Fase de Almacenamiento Se determinan los valores que tendrán los pesos para almacenar un conjunto de patrones (*memorias fundamentales*).

1. Conjunto de patrones que se desea almacenar: $\{\mathbf{x}(k) = (x_1(k), x_2(k), \dots, x_n(k))\}_{k=1, \dots, p}$.
2. Aplicando aprendizaje Hebbiano:

$$w_{ji} = \frac{1}{N} \sum_{k=1}^p x_j(k) x_i(k)$$

La matriz resultante **W** es simétrica y, cómo las neuronas no tiene *self-feed-back*, la diagonal es de ceros.

2) Fase de Recuperación Mecanismo para recuperar la información almacenada a partir de información incompleta.

1. Dado un patrón **x** (incompleto), se fuerza que la salida de la red sea: $\mathbf{y}(0) = \mathbf{x}$.
2. $j^* = \text{rand}(N)$.
3. $y_{j^*}(n) = \text{sign} \left(\sum_{i=1}^N w_{ji} y_i(n-1) \right)$.
4. Volver a 2 hasta no observar cambios en y_{j^*} .

9.3.3. Estados espurios

Todos los posibles estados en una red de Hopfield forman un cubo unitario N -dimensional.

Las M memorias fundamentales (vectores), generan un subespacio constituido por un conjunto de puntos fijos (**estados estables**) representados por ciertas esquinas en el cubo unitario. Las otras esquinas que están dentro, o cerca del subespacio son ubicaciones **potenciales** de *estados espurios*. Estos son estados estables que **no representan** ninguna memoria fundamental.

En el diseño de una red de Hopfield existe un intercambio entre (1) la necesidad de conservar las memorias fundamentales como puntos fijos en el espacio de estados y (2) el deseo de que haya pocos estados espurios.

9.3.4. Capacidad de almacenamiento

Existen dos fenómenos que hacen decrecer la eficiencia de la red de Hopfield:

1. Que las memorias fundamentales no siempre son estables. Serán estables si y solo si la SNR es alta.
2. Estados espurios, que representan otros estados estables distintos de las memorias fundamentales pueden aparecer.

Utilizando métodos probabilísticos podemos definir una **capacidad de almacenamiento**, con un 1 % de error, como el mayor número de memorias fundamentales que pueden ser almacenadas en la red y recuperadas correctamente:

$$p_{\text{máx}} = \frac{N}{2 \cdot \log(N)},$$

dónde $p_{\text{máx}}$ es la cantidad máxima de memorias fundamentales.

9.3.5. Generalidades

- Cada neurona tiene un disparo probabilístico.
- Conexiones simétricas.
- El entrenamiento es no-supervisado.
- Cada patron es un “valle” de energía.
- Se busca el valle iterativamente a partir de ciertas condiciones iniciales.
- El proceso de **entrenamiento** es NO iterativo (lo hace una sólo vez).
- El proceso de **recuperación** si ES iterativo (dinámico).
- La cantidad de memorias fundamentales debe ser poca en relación a la cantidad de neuronas de la red para que los patrones puedan ser recuperados correctamente.

9.4. Arquitecturas de RR

Todas las RR que se listan a continuación comparten la característica de utilizar un MLP *estático* (o partes de él), aprovechándose de su capacidad de no-linealidad.

9.4.1. Modelo NARX

Sigue la estructura de un MLP, pero las entradas al mismo se dividen en dos partes:

- Datos de entrada presentes y pasados: $u(n), u(n-1), \dots, u(n-q+1)$.
- Valores de salidas presente y pasados que son retro-alimentadas al MLP: $y(n), y(n-1), \dots, y(n-q+1)$.

Dónde q es el número máximo de unidades de retraso del modelo. Es entonces que la salida del MLP es función de estas entradas, de forma que:

$$y(n+1) = F(y(n), \dots, y(n-q+1), u(n), \dots, u(n-q+1)).$$

9.4.2. Modelo de Espacio de Estados

Se comporta como un MLP de una sola *capa oculta*. La misma es no-lineal, mientras que la *capa de salida* es lineal.

Los patrones de entrada pasan por la *capa oculta* y son retroalimentados a la misma, antes de dirigirse ambos valores (tanto el patrón procesado como el patrón procesado retroalimentado) a la *capa de salida*.

9.4.3. MLP recurrente (RMLP)

Es un MLP que posee retroalimentación en cada capa de su estructura. El RMLP es un caso particular de la red Elman.

9.4.4. Redes de Segundo Orden

Básicamente, se comportan como un MLP que poseen una sola *capa oculta* llamada *multiplicadores*, cuya cantidad de neuronas en esa capa es igual a la cantidad de entradas por la cantidad de neuronas en la *capa de salida*.

Teniendo x entradas, y y salidas, cada entrada va a conectarse a y neuronas ocultas, pero ninguna entrada se conectará a la misma neurona que otra entrada. Lo mismo sucede con la retroalimentación de las salidas, que se conectarán a x neuronas ocultas.

9.5. Redes Dinámicas de Elman y Jordan

Las redes **Elman** y **Jordan** son redes **parcialmente recurrentes**.

La arquitectura básica en ambas es la de una red MLP. Sin embargo, de todas las entradas de las que se dispone en la capa de entrada, algunas son utilizadas para recoger la información de la que disponían otras neuronas en el instante anterior. Estas neuronas se denominan **entradas de contexto**, dado que hacen referencia al estado anterior de la red.

La diferencia principal entre los dos tipos de arquitectura está en la información que se transmite a las *entradas de contexto*. En las redes **Elman**, se realimentan las salidas de las neuronas de la última capa oculta, de modo que, en cierto sentido, la red dispone de información acerca de la entrada del instante anterior.

En las redes **Jordan**, sin embargo, existe una doble recurrencia. Por una parte se realimentan las salidas del instante anterior ponderadas con un parámetro fijo μ , y además, cada neurona de contexto recibe una copia de su estado anterior. El parámetro μ determina el horizonte de la memoria de la red, es decir, determina la ventana de tiempo que recuerda la red los datos de salida.

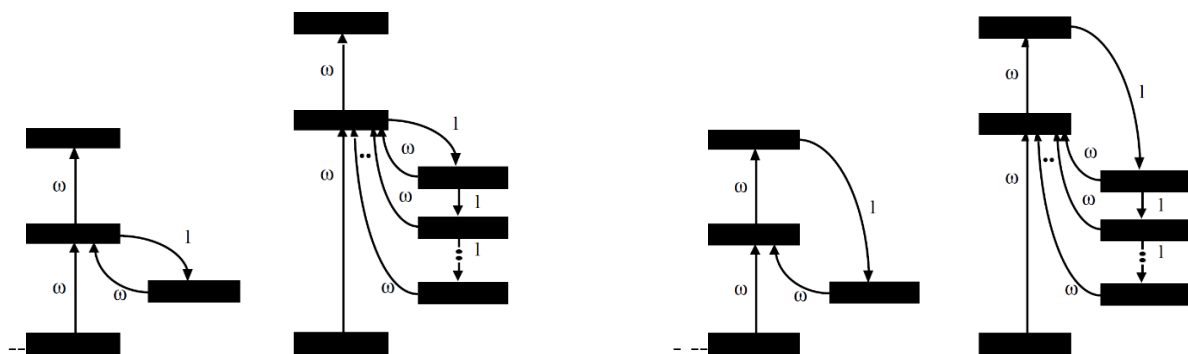


Figura 12: Arquitectura de una red Elman (*izquierda*) y una red Jordan (*derecha*).

El aprendizaje de estos dos tipos de redes se basa en el *algoritmo de retropropagación a través del tiempo*, dado que a la hora de entrenar se desacoplan los bucles. De este modo, inicialmente se calculan las salidas y se hallan los datos para pasar a las neuronas de contexto. En el siguiente instante, se consideran estos datos como entradas a la red, y se aplica de nuevo el algoritmo, y así sucesivamente.

9.6. Retropropagación a través del tiempo

El *algoritmo de retropropagación a través del tiempo* (BPTT) es una derivación del algoritmo de retropropagación tradicional.

Lo que hace es desplegar una RR en una red *feed-forward*, donde cada capa de esta última va a estar construída a partir del conjunto completo de las neuronas de la RR en el instante n : es decir, se va a ir construyendo la topología de a una capa por cada iteración.

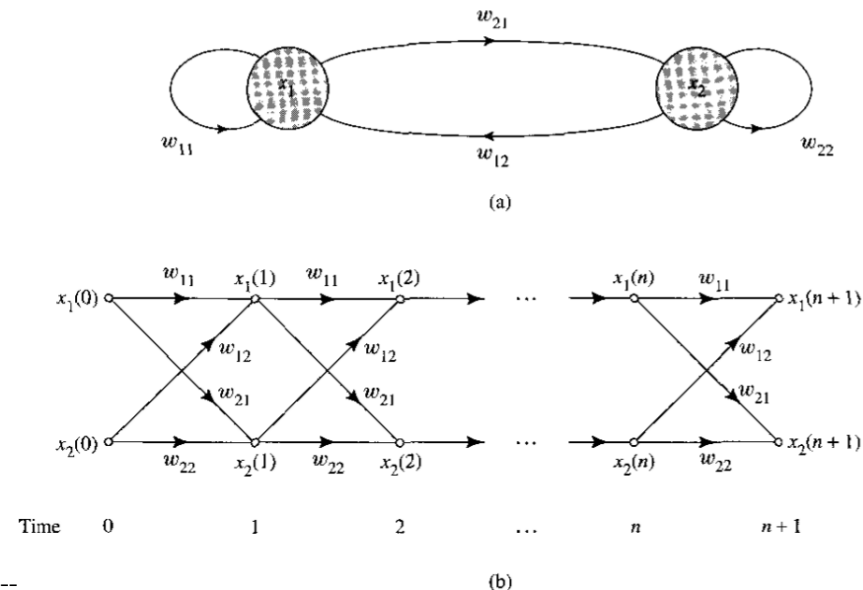


Figura 13: (a) Arquitectura de una RR compuesta de dos neuronas. (b) Gráfico del flujo de la señal utilizando BPTT en la RR de (a).

Existen dos modos de este algoritmo, **por épocas** (hago toda la propagación *feed-forward*, luego toda la propagación *feed-back*, y luego actualizo los pesos) y **truncado** (voy ajustando los pesos a medida que avanzo, considerando una “*profundidad de truncamiento*” que me indica cuantas unidades de tiempo anteriores he de tener en cuenta para los cálculos).

El procedimiento de *despliegue* funciona perfectamente para RR relativamente simples. Sin embargo, las fórmulas se vuelven inmanejables cuando se aplica el algoritmo a arquitecturas más complejas, siendo esta su **mayor debilidad**.

Parte II

Lógica borrosa

1. Lógica

Las **lógicas** son **lenguajes formales** para representar información y/o conocimiento de una forma que sea **tratable por computadoras**.

Tipo de Lógica	Qué Permite Representar	Estados del Conocimiento
Proposicional	Hechos	Verdadero/Falso
De predicados de 1er. Orden	Hechos, objetos, relaciones	Verdadero/Falso
Temporal	Hechos, objetos, relaciones, tiempo	Verdadero/Falso
Probabilística	Hechos	Grado de certeza: $[0, \dots, 1]$
Borrosa	Grado de verdad	Grado de certeza: $[0, \dots, 1]$

Sintaxis: define cómo deben ser las oraciones en el lenguaje.

Semántica: define el significado de las oraciones (por ejemplo: define la veracidad de una oración).

1.1. Lógica Proposicional

La validez de los hechos o proposiciones puede probarse mediante **tablas de verdad** o a través de **Reglas de Inferencia**, las cuales se basan en la propiedad de **monotonidad**:

«*Monotonidad*: las conclusiones se mantienen ante cualquier incremento de premisas; es decir, si tengo una cierta cantidad de hechos verdaderos, y agrego más hechos verdaderos, estos últimos no deben **invalidar** los anteriores.»

Inferencia. Relación entre un conjunto de proposiciones y una proposición tal que la última, denominada **conclusión**, se obtiene necesariamente del primer conjunto, cuyos elementos se llaman **premisas**. Esta relación lógica cumple la propiedad de **monotonidad**.

Supongamos que una **Base de Conocimiento (BC)** justifica un conjunto de afirmaciones. Una lógica es **monótona** si cuando agregamos nuevas afirmaciones a la **BC**, todas las afirmaciones implicadas originalmente siguen siendo implicadas por la **BC** ampliada.

$$\text{Si } BC1 \models a \Rightarrow (BC1 \wedge BC2) \models a$$

Nota: La expresión $BC1 \models a$ es “en la $BC1$ es verdadero a ” ó “en la $BC1$ inferí a ”.

1.1.1. Reglas de Inferencia

Modus Ponens:

Es la más conocida, y se escribe como:

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

“Si α es verdadero, entonces β también lo es. Se dió α , entonces ocurrió β ”.

Modus Tollens:

Está definida como:

$$\frac{\alpha \Rightarrow \beta, \quad \neg \beta}{\neg \alpha}$$

“Si α es verdadero, entonces β también lo es. No ocurrió β , entonces no se dió α ”.

Eliminación- \wedge :

De una conjunción se puede inferir cualquiera de sus conjuntos:

$$\frac{\alpha \wedge \beta}{\alpha}$$

“Si α **y** β es verdadero, puedo inferir que tanto α como β son verdaderas.”

Resolución Unitaria:

Es la regla que aplica Prolog:

$$\frac{\alpha \vee \beta, \quad \neg\beta}{\alpha}$$

“Si α **ó** β es verdadero, y luego sé que β es falso, entonces α ha de ser verdadera.”

El problema de la **lógica proposicional** es que se requiere de un gran número de proposiciones para expresar problemas (cada elemento del mundo hay que representarlo con un hecho verdadero o falso).

1.2. Lógica de Predicados de Primer Orden

Aparece para solventar los problemas de la **lógica proposicional**. Extiende a esta última al emplear **variables**, **predicados** y **cuantificadores de variables**. A su vez es extendida por la **Lógica de Segundo Orden**. Tiene la capacidad para definir prácticamente a todas las matemáticas.

Con la **lógica de predicados** se pueden representar *relaciones entre objetos* y *relaciones entre relaciones*. A la lógica que sólo permite representar *relaciones entre objetos* se la llama de **primer orden**; la que permite *relaciones entre relaciones*, es de **segundo orden**, y así sucesivamente.

Un importante uso de la *Lógica de Predicados de Primer Orden* (LdPO) es para formalizar la semántica de los lenguajes de programación y para especificar y verificar programas, así como también es usada en muchas aplicación de lógica matemática.

La LdPO da bastante *flexibilidad* para representar prácticamente cualquier problema, de un modo bastante intuitivo y cercano a la realidad (cosa difícil de lograr en la *lógica proposicional*).

1.2.1. Reglas de Inferencia

La LdPO posee las mismas reglas de inferencia que la lógica proposicional, además de la de **Generalización Universal** (que caracteriza la LdPO):

«Si se prueba un predicado para un elemento del dominio y se demuestra verdadero, siendo este elemento totalmente arbitrario (o sea, no se conoce nada especial sobre el elemento), entonces se puede generalizar (o afirmar) ese predicado para todos los elementos del dominio.»

1.3. Sistemas de producción con encadenamiento hacia adelante

O también llamados **Sistemas Expertos** (SE). Se comienza desde los **hechos** (*sentencias atómicas*) de la *memoria de trabajo* y se infiere añadiendo los hechos nuevos hasta que no se puedan realizar más inferencias o hasta que el objetivo haya sido agregado.

Cada inferencia es la aplicación de *Modus Ponens*: $P \rightarrow Q, P \models Q$.

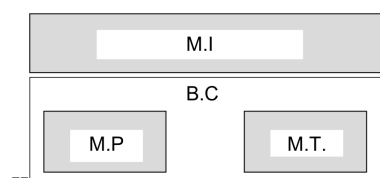


Figura 14: Componentes del SE.

1.3.1. Componentes:

Máquina de Inferencia (MI): es el SE de por sí.

Base de Conocimiento (BC): representa el estado del conocimiento del sistema. Está conformado por la MT y la MP.

Memoria de Trabajo (MT): Contiene un conjunto de literales positivas que no contienen variables (ej: *perro tiene pelo*). Son los hechos que se saben que son verdaderos.

Memoria de Producciones (MP): está constituidas por reglas del tipo:

if <cond 1><cond 2>...<cond n> then <acc 1><acc 2>...<acc n>

+ *Lado izquierdo* (antecedentes): puede contener variables, debe aparear con una afirmación.

+ *Lado derecho* (consecuentes): especifica acciones sobre la MT.

1.3.2. Fases:

1. **Match (fase de cotejo):** Se compara cada *antecedente* con el contenido de la MT. Se incorporan al **conjunto de conflicto** aquellas reglas cuyos *antecedentes* se satisfacen con la MT actual.
2. **Resolución de conflictos:** Se decide cuál de las reglas contenidas en el **conjunto de conflicto** se va a ejecutar. Entre los criterios empleados para su selección tenemos:
 - *No duplicación.* Se va a aplicar una regla que agregue conocimiento y que no se halla aplicado en la fase anterior.
 - *Novedad.* Se va a aplicar una regla que tenga como antecedente conocimiento que fue agregado recientemente a la BC.
 - *Especificidad.* Utilizo la regla que tenga más antecedentes (es decir, la más específica).
 - *Prioridad de operación.* Cuando ninguno de los criterios anteriores se aplica, utilizo un sistema de prioridades para asignar la regla.
3. **Aplicación:** Se aplica el **consecuente** de la regla seleccionada para **todos** los antecedentes que cumplan la regla. Existen dos interpretaciones posibles:
 - a) Se agregan a la MT los hechos que componen el *consecuente*.
 - b) Se ejecutan las acciones indicadas en el *consecuente*.

2. Lógica Borrosa

2.1. Introducción

2.1.1. Motivación

La idea es imitar el comportamiento humano que a veces trabaja con conceptos no del todo certeros. Es decir, existe un cierto grado de incertidumbre.

El ser humano también utiliza **reglas lingüísticas** para comunicarse que no están definidas con precisión numérica. Por ejemplo: “Hace mucho calor” ó “la temperatura es **alta**”.

Incerteza VS Aleatoriedad:

El concepto de **incerteza** es diferente al de **aleatoriedad**.

Aleatoriedad: Se basa en las probabilidades que existen de que ocurra algo tras muchos eventos. Por ejemplo, si se tira cien veces un dado, se puede ver que existe una probabilidad del tanto % de que salga un cierto número. Luego que el dado es arrojado, ya no existe *incerteza* acerca del valor que salió.

Borrosidad: Se relaciona con un solo evento. Por ejemplo, tengo un dado medio redondeado en el que no puedo saber con certeza si el número que salió al tirarlo es 1 o 2. Incluso después de haber sido arrojado, la *incerteza* está ahí.

La **borrosidad** describe entonces la **ambigüedad de un evento**, no mide si se da o no (eso es *aleatoriedad*), mide en **qué medida**, o en **qué grado el evento** se da.

Estos son los conceptos **borrosos** que se desean utilizar.

2.1.2. Conjuntos borrosos y binarios

En un **conjunto binario**, un elemento pertenece o no a uno o más conjuntos. Por otro lado, en los **conjuntos borrosos**, cada elemento pertenece en un **cierto grado** a cada conjunto. Por ejemplo, en la **Figura 15**, el elemento $\{\text{vel} = 30 \text{ km/h}\}$ del conjunto borroso pertenece en un grado de 0.6 al conjunto de las *velocidades lentas* y en un grado de 0.3 al conjunto de las *velocidades medias*.

Es decir, cada elemento del conjunto borroso tiene una **función de membresía** que indica en *qué grado* un elemento pertenece a el.

Consideraciones:

Para cuestiones de análisis haremos una simplificación y analizaremos conjuntos borrosos discretos de dos elementos en \mathbb{R}^2 . De esta manera podremos hacer gráficos que nos permitirán representarlos.

El conjunto a considerar tendrá los elementos P y Q . Cada punto en el hipercubo (de dos dimensiones en este caso) representa un conjunto borroso. Los vértices son casos particulares (“no-borrosos”). A partir de la **Figura 16**, determinamos los siguientes conjuntos:

- Z : Representa el **conjunto vacío**.
- P : Representa el conjunto donde P pertenece totalmente y Q no pertenece.
- Q : Representa el conjunto donde Q pertenece totalmente y P no pertenece.
- X : Es el **conjunto universal** (pertenecen ambos).
- $?$: Es el **conjunto borroso medio**.

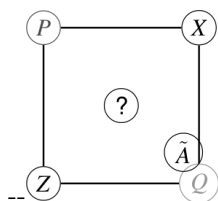


Figura 16: Representación gráfica del CB.

El resto de los puntos (*no-vértices*) son todos *conjuntos borrosos* donde los elementos P y Q pertenecen en cierto grado según su membresía.

Las funciones de membresía para conjuntos tradicionales *binarios* solo pueden tomar valor 0 ó 1. Sin embargo, para conjuntos borrosos, pueden tomar valores continuos entre 0 y 1.

Nota: Esto puede generalizarse aumentándose la dimensión. En el caso continuo se habla de un hipercubo en \mathbb{R}^∞ .

2.2. Operaciones básicas

Sea E un conjunto enumerable y x un elemento de E . Un subconjunto borroso \tilde{A} de E es el conjunto de pares ordenados:

$$\tilde{A} = \{(x_i, \mu_A(x_i))\}; \quad x_i \in E,$$

dónde $\mu_A(x_i)$ es el **grado de membresía** de x_i en A .

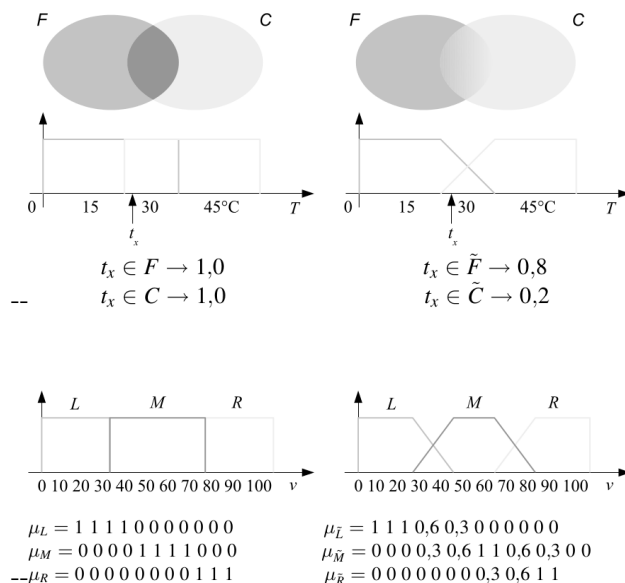
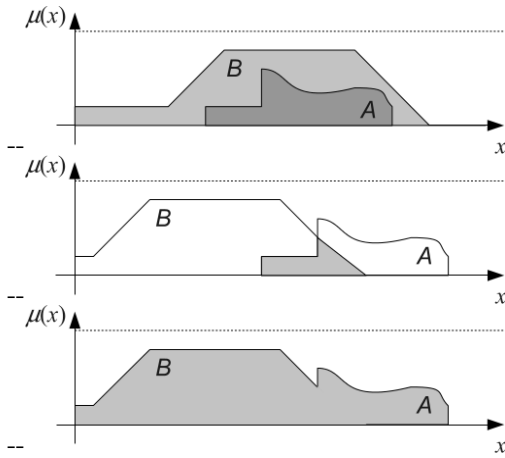


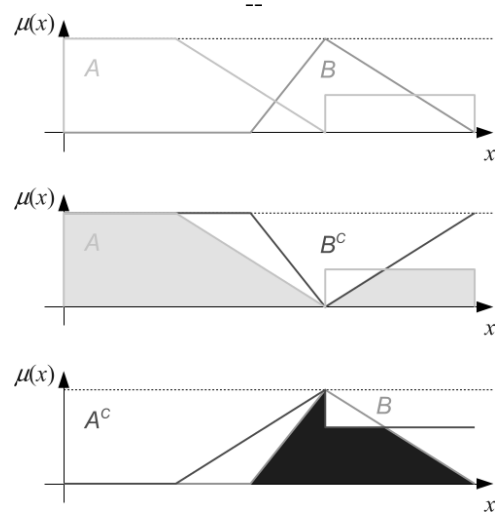
Figura 15: *Arriba:* ejemplo de caso **continuo**. *Izquierda:* conjunto binario. Una temperatura de t_x pertenece a ambos conjuntos. *Derecha:* conjunto borroso. Una temperatura de t_x pertenece en un *grado* a cada conjunto. *Abajo:* ejemplo de caso **discreto**.

2.2.1. Operaciones elementales

1. **Inclusión:** $\tilde{A} \subset \tilde{B} \iff \mu_A(x_i) \leq \mu_B(x_i), \quad \forall x_i \in E.$
2. **Igualdad:** $\tilde{A} = \tilde{B} \iff \mu_A(x_i) = \mu_B(x_i), \quad \forall x_i \in E.$
3. **Complemento:** $\tilde{A} = \tilde{B}^c \iff \mu_A(x_i) = 1 - \mu_B(x_i), \quad \forall x_i \in E.$
4. **Intersección:** $\tilde{A} \cap \tilde{B} \iff \mu_{A \cap B}(x_i) = \min\{\mu_A(x_i), \mu_B(x_i)\} \quad \forall x_i \in E.$
5. **Unión:** $\tilde{A} \cup \tilde{B} \iff \mu_{A \cup B}(x_i) = \max\{\mu_A(x_i), \mu_B(x_i)\} \quad \forall x_i \in E.$
6. **Suma Disyuntiva:** $\tilde{A} \oplus \tilde{B} = (\tilde{A} \cap \tilde{B}^c) \cup (\tilde{A}^c \cap \tilde{B}).$
7. **Diferencia:** $\tilde{A} - \tilde{B} = \tilde{A} \cap \tilde{B}^c.$



(a) Representación gráfica de operaciones elementales: *inclusión, intersección y unión.*



(b) Ejemplo de *suma disyuntiva*. Nótese cómo se trabaja con los complementos. El resultado de $\tilde{A} \oplus \tilde{B}$ es la unión de los grises sólidos de las últimas dos figuras.

2.2.2. Distancias borrosas

Las distancias dan una medida de que tan parecidos son entre sí dos conjuntos borrosos.

Las distancias *relativas* (o *normalizadas*) se utilizan para comparar distancias entre sí cuando los conjuntos tienen distintas cantidades de elementos. Por ejemplo, para ver si los conjuntos borrosos \tilde{A} y \tilde{B} están más separados que el de \tilde{C} y \tilde{D} , independiente de la cantidad de elementos que tengan.

Distancia de Hamming (d), y distancia de Hamming Relativa (δ):

$$d(\tilde{A}, \tilde{B}) = \sum_{i=1}^n |\mu_A(x_i) - \mu_B(x_i)|, \quad \delta(\tilde{A}, \tilde{B}) = \frac{d(\tilde{A}, \tilde{B})}{n}.$$

Distancia Euclídea (e), y distancia Euclídea Relativa (ϵ):

$$e(\tilde{A}, \tilde{B}) = \sqrt{\sum_{i=1}^n [\mu_A(x_i) - \mu_B(x_i)]^2}, \quad \epsilon(\tilde{A}, \tilde{B}) = \frac{e(\tilde{A}, \tilde{B})}{\sqrt{n}}.$$

Nota: En caso de trabajar con casos continuos, se utilizan integrales en vez de sumatorias.

2.3. Caracterización de los conjuntos borrosos

Sirve para resolver la pregunta “¿qué tan borrosos son los conjuntos y cómo se puede medir esa borrosidad en la práctica?”.

2.3.1. Conjunto binario de nivel α

El conjunto binario de nivel α (A_α), asociado al conjunto borroso \tilde{A} es el conjunto binario definido por:

$$A_\alpha = \{x / \mu_A(x) \geq \alpha, \quad \forall x \in A\}.$$

Es decir, convierte el conjunto borroso a un conjunto binario mediante un umbral α . Esta definición es útil para definir el **conjunto convexo**.

2.3.2. Conjunto convexo

Un conjunto es *convexo* si la membresía entre todos los valores intermedios entre dos puntos es mayor o igual que la membresía mínima de esos dos puntos. Matemáticamente:

$$\mu_A(\lambda x_1 + (1 - \lambda) x_2) \geq \min\{\mu_A(x_1), \mu_A(x_2)\}, \quad \forall \lambda \in [0, 1], \forall x_1, x_2 \in \mathbb{R}.$$

La **intersección** de dos conjuntos convexos forman otro conjunto convexo.

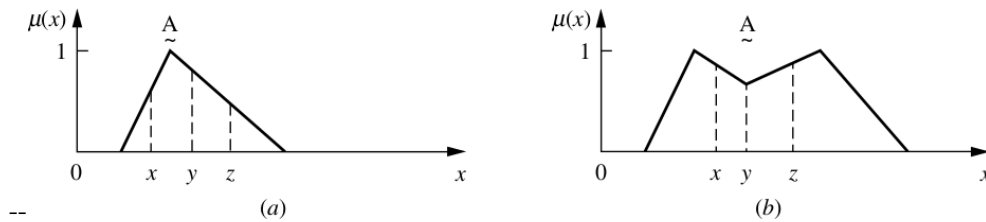


Figura 17: Conjunto borroso: (a) *convexo*; (b) *no-convexo*. Nota: $x = x_1$, $y = \lambda x_1 + (1 - \lambda) x_2$, $z = x_2$.

Se cumple que: \tilde{A} es **convexo** $\iff A_\alpha$ es **convexo** $\forall \alpha \in [0, 1]$.

Esta propiedad es más práctica para determinar conjuntos convexos ya que es más fácil determinar si es *convexo* un conjunto binario que uno borroso.

2.3.3. Conjunto normal

Un conjunto es *normal* cuando su función de membresía tiene al menos un elemento x en el universo de esa membresía cuyo valor sea uno. Es decir:

$$\max\{\mu_A(x)\} = 1, \quad \forall x \in A.$$

2.3.4. Cardinalidad de un conjunto

La *cardinalidad* o *tamaño* de un conjunto es la sumatoria (o integral, en el caso continuo) de las membresías de todos los elementos del conjunto:

$$|\tilde{A}| = \sum_{i=1}^n \mu_A(x_i).$$

2.3.5. Propiedades a tener en cuenta

Hay ciertas propiedades que con los conjuntos **borrosos** hay que tener cuidado, pues se empiezan a no cumplir. Por ejemplo, la **intersección** entre un conjunto y su complemento ya no es más el **conjunto vacío**. Tampoco la **unión** entre un conjunto y su complemento es el **conjunto universal**.

Esto es por cómo están definidas las operaciones elementales.

2.3.6. El conjunto borroso medio

Es el conjunto “*más borroso*”. Tiene a todos los elementos por la mitad, es decir, con membresía de $\frac{1}{2}$:

$$\tilde{M} = \{x / \mu_M(x) = \frac{1}{2}, \quad \forall x \in E\}.$$

Se cumple que es igual a su complemento, y a la intersección y/o unión con este:

$$\tilde{M} = \tilde{M} \cap \tilde{M}^c = \tilde{M} \cup \tilde{M}^c = \tilde{M}^c.$$

2.3.7. Entropía borrosa

La *entropía borrosa*, es una manera de medir la borrosidad de un conjunto borroso \tilde{A} . Se define como el cociente entre la *distancia de Hamming* desde \tilde{A} hasta el conjunto binario (vértice) más cercano y la *distancia de Hamming* entre \tilde{A} y el conjunto binario (vértice) más lejano.

$$S(\tilde{A}) = \frac{d(\tilde{A}, I_{\min}^n)}{d(\tilde{A}, I_{\max}^n)} = \frac{d_{\min}}{d_{\max}}, \quad \begin{aligned} I^n &= \{0, 1\}^n \quad \text{con } n \text{ la cantidad de elementos,} \\ I_{\min}^n &= I_{j^*}^n \iff d(\tilde{A}, I_{j^*}^n) < d(\tilde{A}, I_j^n), \quad \forall j \neq j^*, \\ I_{\max}^n &= I_{i^*}^n \iff d(\tilde{A}, I_{i^*}^n) > d(\tilde{A}, I_i^n), \quad \forall i \neq i^*. \end{aligned}$$

Este método de medición es difícil de utilizar cuando se está en \mathbb{R}^n para n grande. Sin embargo, existe un teorema que da un método para calcularlo de una manera más sencilla: el **teorema de la entropía borrosa**.

Ejemplo del cálculo de entropía:

En base a la **Figura 18**, tenemos que $\tilde{A} = (0,7;0,2)$, con $n = 2$, ya que tenemos dos elementos en los conjuntos que vamos a trabajar.

$$S(\tilde{A}) = \frac{d_{\min}}{d_{\max}} = \frac{|0,7 - 1| + |0,2 - 0|}{|0,7 - 0| + |0,2 - 1|} = \frac{0,5}{1,5} = \frac{1}{3}.$$

Otros valores de entropía de interés son:

$$\begin{aligned} \text{Conjunto borroso medio:} \quad S(\tilde{M}) &= 1. \\ \text{Conjunto binario:} \quad S(\tilde{I}^n) &= 0. \end{aligned}$$

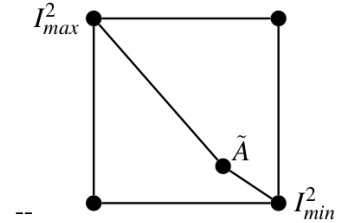


Figura 18: Ejemplo de cálculo de entropía.

2.3.8. Teorema de la entropía borrosa

Este teorema propone una forma más sencilla de calcular la entropía, haciendo:

$$S(\tilde{A}) = \frac{|\tilde{A} \cap \tilde{A}^c|}{|\tilde{A} \cup \tilde{A}^c|}.$$

Se puede demostrar este teorema gráficamente viendo la **Figura 19**. Nótese que las líneas que unen \tilde{A} con I_{\min}^2 es del mismo tamaño que la que une a $\tilde{A} \cap \tilde{A}^c$ con el conjunto binario que tomamos como origen ($I^2 = \{0, 0\}$). Lo mismo con el par \tilde{A}^c y $\tilde{A} \cup \tilde{A}^c$. Por lo que el cociente de utilizar cualquiera de las dos distancias es el mismo.

Nota: las líneas **no** se refieren a la distancia euclídea, sino a la *distancia de Hamming* a los vértices en un caso, y a la *cardinalidad* en el otro.

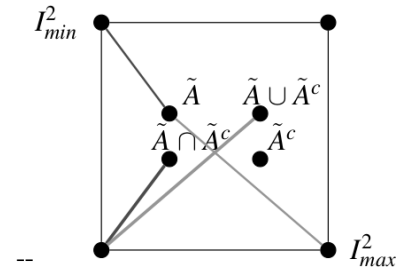


Figura 19: Demostración gráfica del teorema de la entropía borrosa.

2.3.9. Teorema del subconjunto borroso

La “*sub-conjuntez borrosa*” $\phi(\tilde{B}, \tilde{A})$ indica en qué medida \tilde{B} es un **subconjunto borroso** de \tilde{A} . Según el teorema, se calcula como:

$$\phi(\tilde{B}, \tilde{A}) = \frac{|\tilde{A} \cap \tilde{B}|}{|\tilde{B}|}.$$

2.3.10. Teorema de la entropía y el subconjunto borroso

El teorema dice que la *entropía borrosa* de \tilde{A} puede calcularse cómo:

$$\begin{aligned} S(\tilde{A}) &= \phi(\tilde{A} \cup \tilde{A}^c, \tilde{A} \cap \tilde{A}^c), \\ &= \frac{|(\tilde{A} \cap \tilde{A}^c) \cap (\tilde{A} \cup \tilde{A}^c)|}{|\tilde{A} \cup \tilde{A}^c|}. \end{aligned}$$

3. Memorias Asociativas Borrosas

3.1. Introducción

Los conjuntos borrosos son como puntos en el hipercubo $I^n = [0, 1]^n$. Dentro del cubo, la distancia entre puntos miden el tamaño y la borrosidad de los conjuntos borrosos, y permiten medir la cantidad de subconjuntez de un conjunto en otro.

Una **memoria asociativa borrosa** (*fuzzy associative memory*, FAM) es un **sistema borroso** que define un mapeo *entre* cubos borrosos: $S : I^n \rightarrow I^p$. Es decir, las FAMs son **transformaciones** que mapean conjuntos *borrosos* a conjuntos *borrosos*.

Regla Simple (Local FAM):

Es la más simple asociación $(\tilde{A}_i, \tilde{B}_i)$, la cuál asocia un conjunto borroso p -dimensional \tilde{B}_i con un conjunto borroso n -dimensional \tilde{A}_i . Esencialmente, mapean una *pelota* de I^n a una de I^p . Puede leerse condicionalmente como:

$$\text{if } X \text{ is } \tilde{A} \text{ then } Y \text{ is } \tilde{B}.$$

Composición de Reglas (Global FAM):

En general, un **sistema FAM** $F : I^n \rightarrow I^p$ codifica y procesa en paralelo un **banco FAM** de m reglas FAM: $(\tilde{A}_1, \tilde{B}_1), \dots, (\tilde{A}_m, \tilde{B}_m)$. Cada entrada \tilde{A} al *sistema FAM* activa cada una de las reglas FAM almacenadas en diferente grado.

La combinación de cada regla parcial activada \tilde{B}'_i (el ' indica parcialidad) forman la correspondiente salida del conjunto borroso \tilde{B} :

$$\tilde{B} = w_1 \tilde{B}'_1 + \dots + w_m \tilde{B}'_m = \sum_{j=1}^m w_j \tilde{B}'_j,$$

dónde w_j es un peso que refleja la fuerza de la asociación $(\tilde{A}_i, \tilde{B}_i)$.

Nota: Normalmente el valor de w_j es uno, al menos que quiera darle mucho peso a cierta regla particular. Se define por criterios prácticos.

3.2. Fuzzificación

La idea es convertir una entrada no-borrosa x (por ejemplo, un número real o un entero) en un conjunto borroso \tilde{A} (es decir, “*fuzzificarla*”), para poder operar con él y obtener así un conjunto borroso de salida \tilde{B} que será finalmente “*defuzzificado*” para dar con la salida y del tipo original no-borroso.

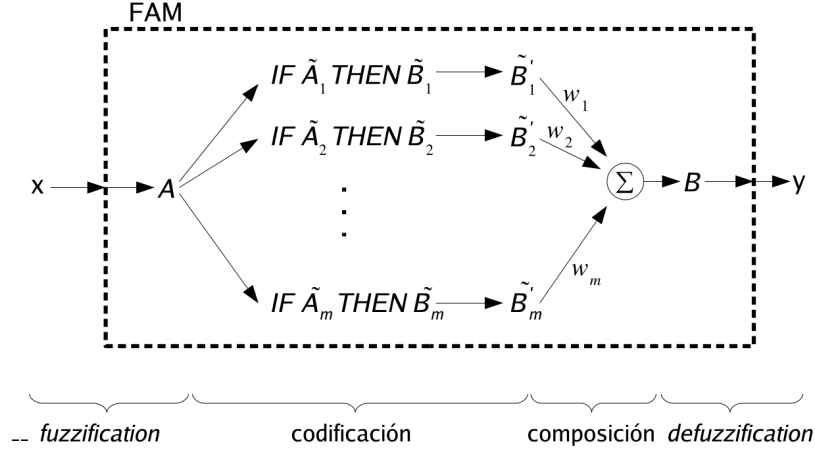


Figura 20: Modelo Aditivo Estándar (SAM).

3.3. Codificación

Una vez determinados los *conjuntos borrosos de entrada* y los *conjuntos borrosos de salida*, procedemos a determinar la codificación de las reglas.

3.3.1. Multiplicación borrosa de Vector-Matrix: Composición máx-mín

El álgebra en el que se basa las FAMs utiliza para multiplicar vectores por matrices lo que se denomina **composición máx-mín**, que usa el operador de composición “ \circ ”. Sean dos vectores \tilde{A} y \tilde{B} y una matriz borrosa de $n \times p$, tal que:

$$\tilde{A} \circ M = \tilde{B} \quad \Rightarrow \quad b_j = \max_{1 \leq i \leq n} \min\{a_i, m_{ij}\}.$$

Es decir, compara punto a punto el vector \tilde{A} de n elementos con cada columna de la matriz M , formando un nuevo vector de mínimos, y luego saca el máximo del vector. Con los p máximos forma el vector \tilde{B} .

Además, existe el operador de **composición máx-producto**, que es definido como:

$$b_j = \max_{1 \leq i \leq n} \{a_i \cdot m_{ij}\}.$$

3.3.2. Fuzzy Hebb's FAMs

La mayoría de los sistemas borrosos utilizados en la práctica son *fuzzy Hebb FAMs*, motivados a partir del aprendizaje hebbiano.

Correlación-mínimo:

Una de las formas para codificar las reglas es utilizando la codificación **correlación-mínimo**, la cuál define la **matriz borrosa Hebbiana** punto a punto como:

$$M = \tilde{A}^T \circ \tilde{B} \quad \Rightarrow \quad m_{ij} = \min\{a_i, b_j\}.$$

La ventaja de esta codificación es que es más barata computacionalmente (porque implementa mín/máx en vez de multiplicaciones/divisiones), pero pierde mucha información por truncamiento.

Se decodifica utilizando la *composición máx-mín*, haciendo: $\tilde{A} \circ M = \tilde{B}$.

Correlación-producto:

Otra forma de codificar las reglas es utilizando la *codificación correlación-producto*. Utiliza el **producto externo** para formar la matriz M :

$$M = \tilde{A}^T \tilde{B} \Rightarrow m_{ij} = a_i \cdot b_j$$

Para la decodificación, se utiliza la *composición máx-producto*, haciendo: $\tilde{A} M = \tilde{B}$.

3.3.3. Teorema de Bidireccionalidad de la correlación-mínimo

Si $M = A^T \circ B$, entonces:

Siendo la **altura** $H(\cdot)$ de un conjunto borroso X :

$$\begin{aligned} (i) \quad A \circ M = B &\iff H(A) \geq H(B), \\ (ii) \quad B \circ M^T = A &\iff H(B) \geq H(A), \\ (iii) \quad A' \circ M \subset B &\quad \forall A', \\ (iv) \quad B' \circ M^T \subset A &\quad \forall B'. \end{aligned} \quad H(X) = \max_{1 \leq i \leq n} \{x_i\}$$

y siendo A' y B' vectores arbitrarios. Si $H(X) = 1$, se dice que es un conjunto borroso **normal**.

En el caso de correlación-producto, se pide en (i) y en (ii) que la altura sea mayor a 1.

3.3.4. Algunas consideraciones prácticas

Si los conjuntos de entrada tienen la forma de un δ_i (por ejemplo, $A = \{0 \ 0 \ 0.75 \ 0 \ 0\}$), se trabajan como conjuntos binarios y se simplifica la matriz:

$$A \circ M_k = A \circ (A_k^T \circ B_k) = (A \circ A_k^T) \circ B_k = a_{ki} B_k$$

dónde k el índice de una regla en particular.

3.4. Composición

3.4.1. Sobreposición de reglas FAM

Puedo integrar todas las m reglas en una sola haciendo:

$$M = \max_{1 \leq k \leq m} M_k.$$

Sin embargo, esto provoca grandes pérdidas de información, debido al truncamiento.

El enfoque borroso es **sobreponer aditivamente los m vectores de B_k** en vez de los M_k , dónde B'_k y M_k son determinados cómo:

$$A \circ M_k = A \circ (A_k^T \circ B_k) = B'_k, \quad \forall A.$$

Esto requiere más espacio para ir guardando separadamente las m asociaciones (A_k, B_k) , pero provee un “*audit-trail*” (como un historial, pero más potente) del procedimiento de inferencia y evita el *crossstalk* (es decir, el usuario puede determinar sin error cuáles reglas incidieron y en cuánto a la formación del conjunto borroso de salida).

A partir de este enfoque, obtenemos el conjunto borroso de salida \tilde{B} utilizando el **modelo aditivo estándar** (SAM):

$$\tilde{B} = \sum_{k=1}^m w_k \tilde{B}'_k$$

o se puede utilizar el modelo SAM con *factor de activación* que es una variante del anterior:

$$\tilde{B} = \sum_{k=1}^m w_k a_k(x) \tilde{B}'_k$$

3.5. Defuzzificación

Ya obtuvimos el conjunto borroso de salida \tilde{B} . Ahora falta calcular la salida no-borrosa y .

El esquema más simple de *defuzzificación* es el de **defuzzificación por máxima-membresía**, el cual elige el y_j con máxima membresía en \tilde{B} como la salida *defuzzificada*:

$$y = \arg \max_j \{\mu_B(y_j)\}$$

Es utilizado cuando el poder de computo es bajo (por ejemplo, en un chip de un lavarropa o un semáforo automático).

Una alternativa que surge, más costosa computacionalmente pero más precisa (siendo esta la más utilizada), es el esquema de **defuzzificación por centroide borroso**:

$$y = \frac{\sum_{j=1}^p y_j \mu_B(y_j)}{\sum_{j=1}^p \mu_B(y_j)}.$$

La salida y es única y usa toda la información de la distribución de \tilde{B} . Este método combina la etapa de *composición* con la de *defuzzificación*.

3.6. Conjuntos de pertenencia continuos

- Los conjuntos son funciones continuas.
- No existen las matrices M (entradas continuas, conjuntos continuos, infinitos puntos en la matriz).
- Método simplificado para guardar los trapecios (considerando que todos los conjuntos tienen forma de trapecio).
- Evaluación de reglas (una a una...).
- Composición de los conjuntos de cada regla y *defuzzificación*.

3.6.1. Cálculo de defuzzificación para los centroides trapezoidales

Para obtener la salida y en los conjuntos continuos, unimos los pasos de *combinación* y *defuzzificación*:

$$y = \frac{\sum_j y_j B_j}{\sum_j B_j} = \frac{\sum \text{centroides} \cdot \text{área de los trapecios}}{\sum \text{área de los trapecios}}$$

Ejemplo:

Siguiendo la **Figura 21**, vamos a calcular y , suponiendo que tenemos sólo dos conjuntos continuos, y que los trapecios son *simétricos*. Además, $c_2 - c_1 = c_4 - c_3$ y $m_2 - m_1 = m_4 - m_3$:

$$\begin{aligned} y &= \frac{y_c \cdot B_c + y_m \cdot B_m}{B_c + B_m} \\ y &= \frac{y_c \cdot a_c \cdot [(c_3 - c_2) + (c_2 - c_1)] + y_m \cdot a_m \cdot [(m_3 - m_2) + (m_2 - m_1)]}{B_c + B_m} \\ y &= \frac{y_c \cdot a_c \cdot (c_3 - c_1) + y_m \cdot a_m \cdot (m_3 - m_1)}{B_c + B_m} \end{aligned}$$

Vemos acá la ventaja de utilizar trapecios, dónde se requiere solamente guardar pocas variables en memoria para trabajar con conjuntos continuos. Y más eficiencia si los trapecios son simétricos.

3.6.2. Múltiples antecedentes por consecuente

Hay varias situaciones en las que se puede tener varios antecedentes, y un solo consecuente.

Para ello se utiliza la **inferencia por descomposición**. La técnica consiste en descomponer por ejemplo la regla compuesta $(A, B; C)$ en dos reglas simples $(A; C)$, $(B; C)$. Estas reglas se procesan en paralelo:

$$\begin{aligned} C_{A'} &= A' \circ M_{AC}, \\ C_{B'} &= B' \circ M_{BC}. \end{aligned}$$

siendo M_{AC} y M_{BC} las matrices que guardan las dos reglas simples.

La idea aquí es recomponer el conjunto borroso C' a partir de la composición de los conjuntos borrosos $C_{A'}$ y $C_{B'}$. Estos se van a componer según como estén los antecedentes unidos: si es por un **and**, se unen utilizando **composición por mínimo**; si lo están por un **or**, se utiliza **composición por máximo**.

En la práctica, se va aplicando un antecedente a la vez, sin necesidad de tener que estar combinando todos los antecedentes en uno.

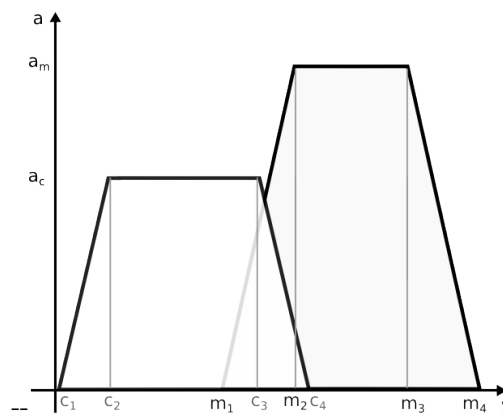


Figura 21: Representación gráfica del método de los centroides trapezoidales.

3.7. Otras alternativas

FAM de entrada/salida binaria (BIOFAM). Caso particular que permite simplificar el problema.

FAM adaptativa. Permite que las reglas se adapten y se creen ellas solas. Funciona de forma similar a una RN. Necesita tener una BD desde la cual va a aprender la FAM. No es la técnica más difundida, ya que normalmente las matrices las arma el ser humano.

3.8. Comparación entre NN vs FS vs ES

	Neuronal Network		Fuzzy System		Expert System
<i>Entradas incompletas o corruptas</i>	Reconstrucción automática.	au-	Reconstrucción automática.	au-	Reglas específicas para considerar esos casos.
<i>Visibilidad del modelo</i>	Caja negra (numérico oculto) ¹ .		Simbólico-numérico visible.	vi-	Simbólico visible.
<i>Estructuración del conocimiento almacenado</i>	No estructurado ² .		Numérico estructurado.		Simbólico altamente estructurado.
<i>Aprendizaje</i>	Automático, gran cantidad de datos.		Experto humano + algoritmo de adaptación ³ .		Experto humano.
<i>Tamaño de la representación</i>	Relativamente pequeña.	pe-	Proporcional a la dimensión del problema.		Crecimiento exponencial de reglas.

Observaciones:

- ¹ : no se puede extraer e interpretar fácilmente el conocimiento observando las componentes de la NN.
- ² : no requiere cambiar la estructura para resolver distintos problemas (cambiarán los pesos y los parámetros, pero *no* la estructura).

- ³ : se puede agregar un algoritmo de adaptación para optimizar el sistema. Puede ser mediante retroalimentación al experto humano, o automáticamente.

3.8.1. Sobre Mamdani, Takagi-Sugeno-Kang y SAM

Los tres tipos principales de sistemas difusos son: Mamdani, Takagi-Sugeno-Kang (TSK) y SAM. El SAM fue desarrollado por Kosko, siendo con TSK modelos similares y aditivos.

La diferencia fundamental entre el modelo de Mamdani y el de SAM/TSK está en los operadores de **composición**, **conjunción** y **disyunción** utilizado en sus razonamientos.

Parte III

Inteligencia Colectiva

1. Resolver problemas mediante búsqueda

1.1. Nociones Básicas

1.1.1. Definición de agente

Un **agente** es cualquier cosa capaz de percibir⁸ su **medioambiente** con la ayuda de **sensores** y actuar en ese medio usando **actuadores** (*reacciona al estímulo ejecutando una acción*).

En general, un agente tomará una decisión en un momento dado dependiendo de la **secuencia completa de percepciones** -*historial completo de lo que ha recibido*- hasta ese instante.

El comportamiento del agente viene dado por la **función del agente** que transforma una percepción dada en una acción. Esta última se implementará mediante un **programa del agente**. La diferencia es que la *función* es una descripción matemática abstracta y el *programa* es la implementación completa.

1.1.2. Diferencia entre agente y programa/objeto

El **agente** se diferencia de un **programa** en lo que respecta a la capacidad de afectar el entorno. El agente cambia el problema, el programa no, ya que este último solo recibe una entrada y devuelve una salida, no ejecuta ninguna acción al respecto. Además, el programa solo se ejecuta una vez, el agente se mantiene continuamente activo.

La diferencia entre un **agente** y un **objeto** (o *clase*), es que el primero puede actuar autónomamente (*puede no responder como se desea*) en base al conocimiento acumulado.

1.1.3. Definición de agente racional

Un **agente racional** es aquel que hace lo **correcto**; es decir, en cada posible secuencia de percepciones, debe emprender aquella acción que supuestamente maximice su **medida de rendimiento**⁹, basándose en las evidencias aportadas por la secuencia de percepciones y en el conocimiento que mantenga almacenado.

Una **acción** es una operación que causa una transición entre estados del mundo.

1.1.4. Clasificación

- **Agentes reactivos.** Basan sus acciones en una aplicación directa desde los estados a las acciones. No funcionan bien en entornos en los que esta aplicación sea demasiado grande para almacenarla y que tarde mucho en aprenderla. Se clasifican a su vez en:
 - **simples**, que no utilizan *historial* para tomar decisiones; ó
 - **basados en modelos**, que mantienen un *estado interno* que depende del *historial* para tomar decisiones.
- **Agentes basados en objetivos.** Pueden tener éxito considerando las acciones futuras y lo deseable de sus resultados.
- **Agente resolvidor de problemas.** Está basado en el anterior; decide qué hacer para encontrar *secuencias de acciones* que conduzcan a los **estados deseables**.

⁸La idea de **percepción** indica que el agente puede recibir entradas en cualquier instante.

⁹Las **medidas de rendimiento** incluyen los criterios que determinan el éxito en el comportamiento del agente. Se prefieren criterios objetivos que estén apuntados a lo que se quiera para el entorno.

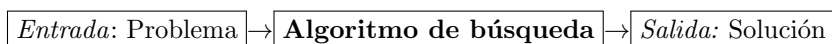
Además están los **agentes basados en utilidad**, que elige decisiones en base a un sistema de prioridad. Por último, los **sistemas multiagentes** contienen varios tipos de agentes, que pueden competir o colaborar para lograr un objetivo.

1.2. Agentes resolutores de problemas

El diseño básico del agente es: *formular, buscar y ejecutar*.

El primer paso para solucionar un problema es la **formulación de un objetivo**, basado en la situación actual y la *medida de rendimiento* del agente. Un **objetivo** es un conjunto de estados del mundo (*estados en los cuales el objetivo se encuentra satisfecho*). Se requiere una forma de medir si se llegó o no al objetivo.

Dado un objetivo, la **formulación del problema** es el proceso de decidir que acciones y estados hay que considerar. Un agente con distintas opciones inmediatas de valores desconocidos puede decidir qué hacer, examinando las diferentes secuencias posibles de acciones que le conduzcan a estados de valores conocidos, y entonces escoger la mejor secuencia. Este proceso de hallar la mejor secuencia se denomina **búsqueda**.



En la última fase, la de **ejecución** se proceden a ejecutar las acciones que recomienda la solución.

1.3. Problemas y soluciones bien definidos

Un problema puede **definirse** mediante cinco componentes:

1. **Estado inicial** en el que comienza el agente.
2. **Conjunto de acciones disponibles**, que pueden incluir información sobre requisitos y consecuencias.
3. **Espacio de estados del problema**, que es el conjunto de todos los estados alcanzables desde el estado inicial por cualquier secuencia de acciones.
4. **Test objetivo**, el cuál determina si un estado es el *estado objetivo*.
5. **Función de evaluación del costo del camino**, que asigna un costo numérico a cada secuencia de acción.

El **proceso de búsqueda** puede ser pensado como la construcción de un **árbol de búsqueda** el cual está superpuesto al *espacio de estados*, donde la raíz corresponde al *estado inicial*. El **estado a expandir** está determinado por la **estrategia de búsqueda**.

1.3.1. Árbol de Búsqueda

Una posible estructura para representar los nodos del árbol es la siguiente:

1. **Estado**: del *espacio de estados* que corresponde al nodo.
2. **Nodo Padre**: el nodo que lo ha generado.
3. **Acción**: que se aplicará al padre para generar el nodo.
4. **Costo del Camino**: desde el estado inicial al nodo.
5. **Profundidad**: que es el número de pasos a lo largo del camino desde el estado inicial.

Nota: un *estado* es una representación de la configuración física, y un *nodo* es una estructura de datos que forma parte del árbol de búsqueda. No son lo mismo.

1.4. Medir el rendimiento de la resolución del problema

Las **estrategias de búsquedas** (que determinan el próximo nodo a expandir) se consideran a partir de los siguientes criterios:

- **Complejidad:** *¿encuentra solución si ésta existe?*
- **Complejidad:** se clasifica en **temporal** (*¿cuánto tiempo tarda?*) y **espacial** (*¿cuánta memoria ocupa?*). Es medida en función de:
 - Máximo factor de ramificación del árbol.
 - Profundidad de la solución más barata.
 - Profundidad máxima del espacio de estado.
- **Optimalidad.** *¿encuentra la solución de mejor calidad entre las disponibles?*

A la vez se clasifican en:

- **Búsqueda no informada.** No tiene información adicional acerca de los estados más allá de los que proporciona la definición del problema. No conoce que tan cerca está del objetivo. Dentro de esta categoría, tenemos los siguientes métodos de búsqueda:
 1. **Horizontal:** construye un árbol que va creciendo a lo ancho. Es **completa** (*al menos encuentra una solución*), y además la solución puede ser **óptima** (*si el costo es uno en cada etapa*). Tiene mucho costo **espacial** (*guarda todo el árbol en memoria*) y **temporal**. Es una buena alternativa si esto último no es problema.
 2. **Profundidad:** construye el árbol yendo hacia abajo. Si la solución no la encuentra, retrocede y desciende por otra rama. No es **completa** (*falla en espacios infinitos y con bucles de estados repetidos*) ni da la solución **óptima**. Es muy **rápido** y requiere **poco espacio** (*necesita almacenar sólo un camino desde la raíz a un nodo hoja, junto con los nodos hermanos restantes no expandidos para cada nodo del camino*).
 3. **Profundidad Iterativa:** es como la anterior, pero iterativamente fija profundidades incrementales. Esta alternativa permite que el algoritmo sea **completo** y **óptimo** (*si el costo es uno en cada etapa*). Es **rápido** y **liviano**.
 4. **Costo Uniforme:** va expandiendo los nodos y calculan los costos, eligiendo siempre los nodos con menor costo. El problema es como asignar costos a los caminos. Es **completo** y **óptimo**. El **tiempo** y el **espacio** que use puede ser alto, dependiendo esto mucho de la función que utilice para calcular los costos.
 5. **Bidireccional:** expande los nodos desde el *estado inicial* con una búsqueda hacia delante y desde el *estado final* con una búsqueda hacia atrás (*requiere necesidad de poder calcularse nodos predecesores*). Requiere definir los algoritmos de búsqueda usados en ambos sentidos. Es **completa** y **óptima** (*si la búsqueda utilizada es la horizontal*). Es muy costosa en **tiempo** y **espacio** (*requiero dos árboles de búsqueda*).
- **Búsqueda informada.** Saben si un estado no-objetivo es “*más prometedor*” que otro. Lo que se hace es aplicar una *función de evaluación* (que suele ser una *estimación*) en el nodo, para ir midiendo el avance. En estos métodos se expande siempre el “*mejor*” nodo (**método de búsqueda primero el mejor**). Existen dos enfoques:
 1. **Avara:** utiliza solamente la heurística (*cuánto me falta para llegar*) para tomar las decisiones. Siempre expando el nodo que se estima más cercano al objetivo. No es **completa** (*pueden existir bucles*) y no es **óptima**. Es bastante costosa en **tiempo** y **espacio**.
 2. **A*:** combina la heurística con el *costo (desde el nodo inicial)* para tomar las decisiones. Es **óptima** y **completa**, si se aplica la restricción de que el mínimo costo estimado al objetivo no se sobre-estime. La elección de una buena función heurística es necesaria para disminuir la complejidad tanto en **tiempo** como en **espacio**. Está dentro de los mejores métodos de búsqueda.

2. Planificación

La **planificación** es el proceso de *búsqueda y articulación* de una **secuencia de acciones** que permitan alcanzar un objetivo.

2.1. Lenguaje de los problemas de planificación

Plantear el problema de búsqueda usualmente exige demasiadas **acciones** y **estados** para analizar.

Algunas de las características que requiere cumplir el problema para ser resuelto con planificación:

- Representación **explícita** del objetivo, para evitar ser desbordado por acciones irrelevantes al problema. Se ha de encontrar una **función heurística**¹⁰ adecuada (para que los algoritmos de búsqueda sean eficientes). El tomar decisiones obvias o importantes en forma temprana permite reducir el **factor de ramificación**, disminuyendo la necesidad de *back-tracking*.
- **Descomposición** del problema en sub-problemas (frecuentemente no es posible), aunque después se necesita trabajo adicional para combinar los **sub-planes** resultantes.
- Suposición de **independencia** de objetivos. Si hay algún conflicto, se tienen mecanismos para resolverlo.
- Entornos: **completamente observables** (el agente conoce a todo momento todas las variables del entorno), **determinísticos** (el próximo estado del medio solo depende del actual y de lo que haga el agente), **estáticos** (los cambios los producen los agentes) y **discretos** (en tiempos, acciones, efectos y objetos).

La **representación de los problemas** de planificación (*acciones, estados y objetivos*) debe hacer posible que los algoritmos de planificación se aprovechen de la estructura lógica del problema. El lenguaje de representación básico de los planificadores clásicos se llama **STRIPS**, y la idea de este es que sea suficientemente **expresivo** (para representar una gran gama de problemas) a la vez que **restrictivo** (para permitir algoritmos operativos y eficientes). Algunas de las características de **STRIPS** son:

- **Representación de estados.** Utiliza representaciones completas de los estados. Posee un conjunto de literales *lógicas de primer orden (sin dependencias funcionales)* **positivas** (*sin negaciones*) y **simples**. Se asume **hipótesis de mundo cerrado** (*todas las condiciones no mencionadas son tomadas como falsas*) y **espacio de estados finito**.
- **Representación de objetivos.** Solo se tiene un test para verificar el objetivo y una función heurística.
- **Representación de acciones.** Consisten en funciones que generan descripciones de estados sucesores. Una **acción** se representa a través de: *precondiciones (literales que tienen que ser verdaderos)* y *efectos (que agreguen/saquen literales)*.

La planificación va creando entonces un **plan** y se van incorporando las acciones en cualquier lugar que sean necesarias. No hay necesariamente una conexión entre el **orden** en que se *genera* el plan y el orden en que se *ejecuta*.

2.2. Operadores STRIPS

Un **operador** (instancia de una regla) **STRIPS** se define mediante **esquemas de representación**. La descripción **explícita** de los *operadores* permite proceder con un enfoque **hacia atrás** (*regresión*), debido a que estos tienen suficiente información para **regresar** desde la descripción parcial de un **estado** hasta la descripción parcial del **estado precedente**.

El enfoque de **regresión** es preferible, ya que usualmente, la descripción del objetivo tiene pocos elementos, para los cuales hay pocos operadores que pueden producirlos. Además, se consideran sólo las acciones **relevantes**.

¹⁰Se puede definir **heurística** como un arte, técnica o procedimiento práctico o informal, para resolver problemas.

2.3. Planificador de Orden Parcial

Las búsquedas anteriores son **totalmente ordenadas**: solo exploran secuencias estrictamente lineales de acciones conectadas directamente al inicio o al objetivo. Esto evita sacar provecho de la **descomposición** del problema.

En este enfoque se *flexibiliza* el *orden* en que se **construye** el plan, tomando las decisiones usando un sistema de prioridades.

Características del **orden parcial** (POP):

- El planificador trabaja sobre **sub-metas** en forma independiente una de otra.
- No se preocupa del orden de los pasos durante la búsqueda.
- Estrategia: **mínimo compromiso**, en la que se aplazan las opciones hasta un punto en el que se pueda seguir progresando en la búsqueda.
- Pueden aparecer dos acciones en un plan sin identificar orden entre ellas.

A la solución dada por un POP hay que **linealizarla**. Es decir, ordenar los pasos. Por ende, puede haber muchas linealizaciones de una misma secuencia de acciones (por ejemplo, *poner media izquierda y luego derecha, o al revés*) que den a lugar varios posibles planes de **orden total**.

Para que un plan sea una **solución**, debe ser:

- **Completo**. Un plan es *completo* \iff se han alcanzado todas las **precondiciones**.
 - + Una *precondición* se ha alcanzado \iff es el efecto de un paso anterior y no puede ser removida por otro paso.
- **Consistente**. Para esto, se requiere que:
 - + No haya ciclos en las restricciones ordenadas.
 - + No haya conflictos con los vínculos causales.

3. Computación Evolutiva

La evolución es un proceso de **optimización**, donde el objetivo es mejorar la habilidad de los individuos de sobrevivir. La computación evolutiva (EC) es la emulación del proceso de **selección natural** en el procedimiento de búsqueda. En la naturaleza, los organismos poseen ciertas características que influyen sus habilidades de supervivencia y reproducción. Estas características están representadas por una larga cadena de información contenida en los **cromosomas** del organismo. Luego de la **cruza**, los cromosomas de los descendientes consisten de una combinación de la información de los cromosomas de los padres. Con suerte, el resultado obtenido es aquel que contiene la mejor parte de los cromosomas de ambos padres. El proceso de selección natural se asegura que los organismos que mejor se “*adaptan*” al entorno sean los que más posibilidades tengan de reproducirse, cuyos hijos estarán similarmente o incluso mejores adaptados.

Ocasionalmente, los cromosomas de un organismo son sujetos a **mutaciones** que pueden causar cambios en las características de dicho individuo. Estos cambios pueden tener una connotación negativa en la habilidad de supervivencia o reproducción del individuo, pero por otro lado, puede que esa mutación realmente mejore la adaptación del mismo, llevando de esa forma mayores probabilidades de supervivencia y producción de descendientes. Sin las mutaciones, la población tiende a converger a un estado homogéneo donde los individuos raramente varían unos con otros.

La evolución vía la selección natural que elige individuos de la población aleatoriamente puede ser visto como una búsqueda a través del espacio de todos los valores de cromosomas posibles. En ese sentido, un **algoritmo evolutivo** (*evolutionary algorithm*, EA) es una búsqueda aleatoria por una solución óptima a un problema dado. El proceso de búsqueda evolutiva está influenciado por las siguientes componentes principales:

- una codificación de la solución al problema en forma de cromosoma;

- una función que evalúa la capacidad de adaptación del individuo;
- una inicialización de la población inicial;
- operadores de selección natural; y
- operadores de reproducción.

Los EAs han sido aplicados a una gran amplia cantidad de áreas, incluyendo:

- planificación, como por ejemplo, optimización de rutas y calendarización;
- diseño, como por ejemplo, el diseño de filtros y arquitecturas neuronales; y
- minería de datos.

3.1. Algoritmos genéticos

Los **algoritmos genéticos** (GAs) son la clase más popular de EA, los cuales modelan la **evolución genética**. Las características de los individuos son entonces expresadas usando **genotipos**. Son utilizados frecuentemente en problemas de optimización.

3.2. La evolución como un algoritmo

```
// Se comienza inicializando la población al azar.
1. Inicializar (población)
   // Se decodifica el genotipo en fenotipo y evalúa el fitness de c/individuo.
2. mejor_fitness = Evaluar (población)
   // Entramos en el b ucle de optimizaci n/b squeda.
3. Mientras (mejor_fitness < fitness_requerido)
   // Seleccionamos los padres de la nueva generaci n.
   3.1 selecci n = Seleccionar(poblaci n)
   // Efectuamos las cruzas y las mutaciones.
   3.2 poblaci n = CruzarYMutar(selecci n)
   // Finalmente, la poblaci n nace y es evaluada.
   3.3 mejor_fitness = Evaluar(poblaci n)
4. FinMientras
```

3.3. Elementos de un algoritmo evolutivo

- **Representaci n de los individuos.** Determinar la traducci n **fenotipo** \leftrightarrow **genotipo**. El primer aspecto a resolver es el de codificar el problema en un diccionario finito.
- **Funci n de *fitness*.** Se debe poder medir que tan buena es cada soluci n en relaci n a las dem s. Tiene las siguientes caracter sticas generales:
 - **Monotici dad.** Cuanto mejor la soluci n, m s grande el n mero.
 - **Precisi n.** Depedende de la cantidad de **bits** utilizados por los cromosomas.
 - **Suavidad Regulable.** Alg n par metro que permita graduar la suavidad, seg n el problema.
 - **Penalizaci n de Complejidad.** Adem s de lograr la soluci n  ptima, se desea que sea simple.

- **Mecanismos de selección.** Elegir a los padres siguiendo probabilidades, como en la naturaleza. Se debe dar la posibilidad incluso de que los *peores* individuos sean padres.
- **Operadores de variación y reproducción.** Los operadores básicos son cruza y mutaciones, habiendo varias formas de aplicarlos. A partir de los operadores podemos reproducir y obtener una nueva población.

3.4. Diseño de una solución mediante algoritmos evolutivos

3.4.1. Representación de los individuos

- **Genético:** representación BINARIA (*genotipo*).
 - Muchos genes con pocos alelos: convergencia asegurada por el *teorema de esquemas*.
 - **Epitasis:** un gen incorrecto invalida todo el cromosoma.
 - Representación lejana al dominio del problema.
 - Gran cantidad de soluciones inválidas en la población.
- **Evolutivo:** representación REAL (*fenotipo*).
 - Pocos genes con muchos alelos.
 - Convergencia muy dependiente de operadores.
 - Necesidad de redifinición de operadores “*no biológicos*”.

Nota: Se ha de establecer un compromiso entre la resolución de la codificación y la cantidad de dimensiones del espacio de búsqueda: mientras más grande el cromosoma, más amplio el espacio de búsqueda.

3.4.2. Estrategias de selección

Rueda Ruleta.

$$\left. \begin{array}{ll} \text{Alto } fitness & \Rightarrow \text{ alta porción de ruleta} \\ \text{Bajo } fitness & \Rightarrow \text{ baja porción de ruleta} \end{array} \right\} \text{ con probabilidad de ser elegido } \propto fitness_{\text{individuo}}.$$

Desventajas: **mar de mediocres** (*demasiadas soluciones mediocres toman amplio espacio de la ruleta \rightarrow convergencia lenta*); **mar de buenas soluciones** (*tiende a uniformizar la población*)

Ventanas. Ordena a los individuos por *fitness* y va tomando ventanas de tamaño incremental. La mayor probabilidad de ser padre se asigna a los mejores individuos, ya que aparecen en todas las ventanas de selección.

Competencia. Toma $k > 1$ individuos, los hace competir por *fitness*, y toma al ganador como padre. Es el más **usado** y el más **sencillo**.

3.4.3. Operadores de variación y reproducción

La **reproducción** es el proceso mediante el cual se obtiene la nueva población a partir de individuos seleccionados y los operadores de variación (*cruza simples, mutación*).

Reemplazos durante la reproducción:

Reemplazo total. Todos los individuos son obtenidos a partir de cruza y mutaciones de los padres.

Reemplazo total con *brecha generacional*.¹¹ Transferimos a la nueva población los padres seleccionados, y completamos los individuos faltantes mediante variaciones.

¹¹La *brecha generacional* determinar cuantos padres son copiados directamente a la población nueva. Es un número real entre [0,1]. Se puede ver como el porcentaje de población que será ocupado por los padres.

Elitismo. Se busca el mejor individuo de la población anterior e **independientemente** de la **selección** y **variación**, se lo copia exactamente en la nueva población. De esta manera, no se pierde la mejor solución.

Operadores de variación:

Mutaciones: evitan caer en *mínimos locales*, permitiendo que constantemente se redistribuya la población sobre el espacio de búsqueda. ¿ $p_{\text{mutación}}$ muy alta? \rightarrow se retrasa o impide la convergencia al perder buenas soluciones. Usando **elitismo** se evita que se pierda la mejor solución.

Cruzas: se clasifican en:

- **Simples:** se elige un punto de cruce al azar y se intercambian los *cromosomas*.
- **Múltiples:** se corta el *cromosoma* en más de dos partes para realizar el intercambio.

3.5. Características principales y variantes

3.5.1. Características principales

Los parámetros que controlan la evolución de un algoritmo genético son entonces:

- Probabilidad de **mutaciones/cruzas**.
- **Tamaño** de la población.
- **Brecha generacional**.
- **Elitismo**.

La **selección** natural actúa sobre el *fenotipo* y suele disminuir la diversidad, haciendo que sobrevivan solo los individuos más aptos; los mecanismos que generan diversidad y que combinan características (**mutaciones** y **cruzas**) actúan habitualmente sobre el *genotipo*.

La búsqueda altamente eficiente los GA es explicada por lo que se ha denominado **paralelismo implícito**, un aspecto fundamental de los mismos. El GA evalúa N *cromosomas* (ó soluciones) de forma directa, pero implícitamente evalúa N^3 esquemas, dado que en el caso de una representación *binaria* un cromosoma representa 2^d esquemas diferentes, con d la longitud del cromosoma.

Comparación con otros métodos:

Métodos tradicionales	Algoritmos evolutivos
Trabajan con los propios parámetros a optimizar.	Emplea una codificación de los parámetros.
Utilizan información de las derivadas de la función objetivo u otro conocimiento adicional.	Utilizan la información de la función objetivo en forma directa .
Reglas de transición deterministas .	Reglas de transición probabilísticas .
Explorar el espacio de soluciones a partir de un punto .	Exploran el espacio de soluciones en múltiples puntos a la vez.

3.5.2. Tratamiento de las restricciones del problema

¿Cómo se pueden considerar las restricciones del problema durante la evolución?

- Redefinición de la representación de forma de que siempre se generen fenotipos válidos.
- Rechazo o eliminación de individuos inválidos.
- Reparación del material genético.

- Modificación de los operadores de variación.
- Esquemas de penalización en la función de aptitud.

3.6. Programación Genética

La **programación genética** (GP) es vista como una especialización de los algoritmos genéticos. Similares a los GAs, GP concentra la evolución en los genotipos. La diferencia principal está en el esquema de representación usado. Dónde GAs utilizan representaciones con cadenas, GP representa los individuos como programas en forma de *árboles*. El objetivo de GP es evolucionar programas de computadora. Por cada generación, cada programa evolucionado (individuo) es ejecutado para medir su *performance* dentro del dominio del problema. Los resultados de la *performance* del programa de computadora evolucionado es luego usado para cuantificar el *fitness* de ese programa.

Las **cruzas** se hacen mediante el intercambio de ramas, mientras que las **mutaciones** pueden hacerse mediante reemplazos de ramas, o generaciones al azar de todo o parte del árbol.

4. Enjambres de partículas y colonias de hormigas

Un **enjambre** puede ser definido por una colección *estructurada* de organismos que **interactúan** entre sí para cumplir un **objetivo global**, de una manera más eficiente que si lo hubiesen hecho individualmente; ejemplo de individuos lo son las hormigas, abejas, avispas, peces y pájaros.

Dentro de estos organismos, los individuos son relativamente **simples** en estructura, pero su **comportamiento colectivo** se vuelve bastante **complejo**. Este comportamiento moldea y dicta el comportamiento del enjambre. Por otro lado, el comportamiento del enjambre determina las condiciones en las cuales los individuos realizan acciones, las cuales a su vez pueden cambiar el entorno, y por ende, cambiar el comportamiento de los individuos.

La interacción entre individuos ayuda a refinar el conocimiento experimental acerca del entorno, y mejora el progreso del enjambre hacia la optimización. La interacción o cooperación entre individuos está determinada genéticamente o a través de interacción social.

Una sorprendente consecuencia de las estructuras de redes sociales en los enjambres es su habilidad de **auto-organizarse**¹² para cumplir su *objetivo global* óptimamente.

El modelado computacional de enjambres ha resultado en numerosas aplicaciones con éxito, como: *optimización de funciones, encontrar la ruta óptima, calendarización, optimización estructural, análisis de imágenes e información*.

4.1. Autómatas de estados finitos y autómatas celulares

Los autómatas celulares son sistemas dinámicos discretos cuyos elementos tienen una interacción constante entre sí tanto en el espacio como en el tiempo. Tienen la capacidad de representar comportamientos complejos a partir de una dinámica sencilla.

Autómata de estados finitos:

Definición:

$$A = \langle X, Y, E, D \rangle$$

dónde X = entradas, Y = salidas, E = estados internos, D = reglas de transición (determinísticas, probabilísticas).

Autómata celulares:

Definición:

$$R = \langle A, T, C \rangle$$

dónde T = topologías (triangular, rectangular, ...), C = medios de conexión (tipos y tamaño de vecindad, tipos de conexiones).

¹²La **auto-organización** es un proceso por el cual se forma un orden global o coordinación entre las componentes de un sistema inicialmente desordenado. El proceso es espontáneo: no está controlado por un agente o un subsistema dentro o fuera del sistema de forma directa; pero las reglas seguidas por el proceso y sus condiciones iniciales si han de ser determinadas por un agente. La organización resultante es altamente **descentralizada**, por lo que es muy robusta y puede recibir cierta cantidad de daño antes de degradarse.

4.2. Optimización por Enjambre de Partículas

4.2.1. Inspiración biológica de los métodos de inteligencia colectiva

El intento inicial del concepto de enjambres de partículas fue el de simular el vuelo de las aves, cuyas bandadas vuelan sincrónicamente y cambian súbitamente de dirección, pero reagrupándose de forma óptima.

En la **optimización por enjambres de partículas** (*particle swarm optimization*, PSO), los individuos (partículas) “vuelan” a través de un espacio de búsqueda hiperdimensional. Los cambios en una partícula están influenciados por su experiencia y por la experiencia de sus vecinos. En consecuencia, el proceso de búsqueda es tal que las partículas retornan estocásticamente a las regiones anteriores del espacio de búsqueda que resultasen mejores.

4.2.2. Estructura social

Está determinada por la formación de sus vecindades. Individuos vecinos se comunican entre ellos. Tenemos las siguientes topologías (ver **Figura 22**):

Topología estrella: todos se comunican con todos. Cada partícula es atraída hacia la **mejor solución** de todo el enjambre.

Topología anillo: cada partícula se comunica con n vecinos inmediatos. Cada partícula es atraída hacia la mejor solución de su vecindad.

Topología rueda: solo una partícula se comunica con todas. Solo esta partícula ajusta su posición hacia la mejor solución, y la mejora es comunicada al resto (si hubo).

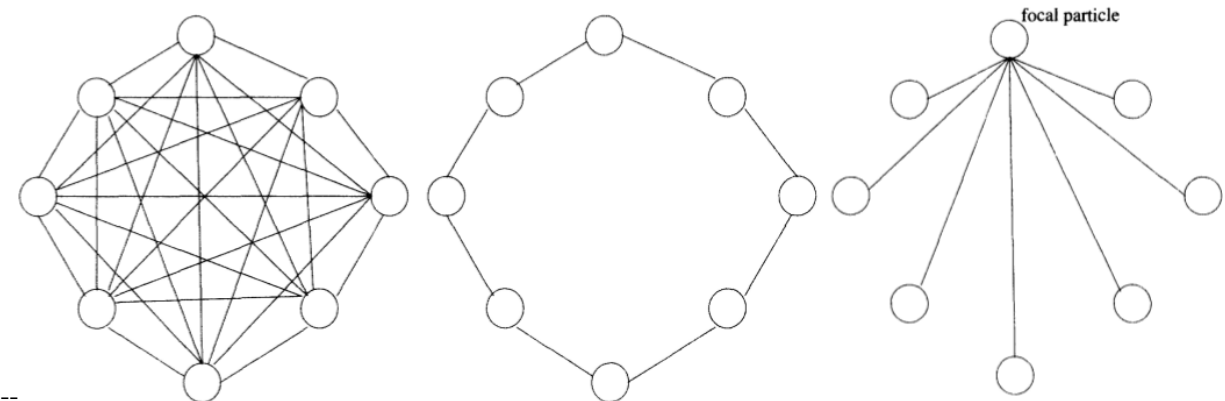


Figura 22: Izquierda: topología estrella; centro: topología anillo; derecha: topología rueda.

Nota: La vecindad se determina según **índices numéricos**, y no por medidas especiales (como la *distancia euclídeana*, por ejemplo).

4.2.3. Algoritmo

Sea $\mathbf{x}_i(t)$ la posición de la partícula P_i en el hiperspacio, en la iteración t . La posición de P_i es modificada por la velocidad $\mathbf{v}_i(t)$ de acuerdo a:

$$\mathbf{x}_i(t) = \mathbf{x}_i(t-1) + \mathbf{v}_i(t). \quad (1)$$

A partir de esto, veremos tres diferentes algoritmos para resolver el problema, que difieren entre sí en la forma en que intercambian información social.

Individual Best (*pbest*):

Versión básica del algoritmo, en que las partículas sólo utilizan sus propias experiencias.

1. Inicializar el enjambre de partículas con valores de posición aleatorios en el hiperespacio.
2. Evalúo la *performance* \mathcal{F} de cada partícula en su posición actual.
3. Realizo la comparación junto a la asignación:

$$\text{if } \mathcal{F}(\mathbf{x}_i(t)) < pbest_i \quad \text{then:} \quad \begin{cases} pbest_i = \mathcal{F}(\mathbf{x}_i(t)), \\ \mathbf{x}_{pbest_i} = \mathbf{x}_i(t). \end{cases}$$

4. Y luego cambiamos la velocidad cada partícula haciendo:

$$\mathbf{v}_i(t) = \mathbf{v}_i(t-1) + \rho(\mathbf{x}_{pbest_i} - \mathbf{x}_i(t)).$$

dónde ρ es un número positivo aleatorio.

5. Movemos a la partícula a la nueva posición (usando 1) y avanzamos una iteración $t = t + 1$.
6. Volver al paso (2), y repetir hasta *convergencia*.

A medida que más se aleja la partícula de su *pbest*, mayores son los cambios en la velocidad para retornar a esa posición. El valor de ρ influye en como la trayectoria de una partícula oscila. Mientras más chico ρ , más suave es la trayectoria.

Global Best (*gbest*)

Refleja la topología **estrella**. La experiencia de cada partícula es compartida hacia el resto, influyendo en los movimientos de estos.

El algoritmo es similar al *pbest*, exceptuando el agregado de un nuevo paso luego del (3), y el cambio en la forma de calcular la velocidad de la partícula:

4. Luego de realizar la comparación con la asignación, se procede a determinar el *gbest*:

$$\text{if } \mathcal{F}(\mathbf{x}_i(t)) < gbest \quad \text{then:} \quad \begin{cases} gbest = \mathcal{F}(\mathbf{x}_i(t)), \\ \mathbf{x}_{gbest} = \mathbf{x}_i(t). \end{cases}$$

5. Y el cambio en la velocidad está dado por:

$$\begin{aligned} \mathbf{v}_i(t) &= \mathbf{v}_i(t-1) + \rho_1(\mathbf{x}_{pbest_i} - \mathbf{x}_i(t)) + \rho_2(\mathbf{x}_{gbest} - \mathbf{x}_i(t)), \\ &= \mathbf{v}_i(t-1) + \text{componente } \textit{cognitiva} + \text{componente } \textit{social}. \end{aligned}$$

dónde se ha incorporado el término de *gbest*, siendo ρ_1, ρ_2 , variables aleatorias.

El resto del algoritmo procede igual que el de *pbest* desde el paso (5).

Mientras más alejado esté la partícula del *gbest* y de su propio *pbest*, el cambio en la velocidad para retornar a su mejor posición será mayor. Las variables ρ_1 y ρ_2 están definidas como $\rho_i = r_i c_i$, con $r_i \sim U(0, 1)$, y c_i una constante positiva de aceleración. Para evitar diverger, $c_1 + c_2 \leq 4$.

Local Best (*lbest*)

Refleja la topología **anillo**. Las partículas son sólo influenciadas por sus propios *pbest*, como así también por la mejor posición de su vecino *lbest*. Del algoritmo *gbest* sólo cambia el paso (4) y (5) al reemplazar *gbest* por *lbest*.

A pesar de ser *lbest* **más lento** en converger que *gbest*, presenta mejores soluciones y busca en un área mucho más grande del espacio de búsqueda.

4.2.4. Convergencia

La convergencia se llega cuando el PSO cumple un cierto número de iteraciones, cuando su mejor *performance* ha cruzado un umbral, o cuando los cambios en las velocidades de las partículas son cercanos a cero.

4.2.5. Parámetros del Sistema

1. **Dimensión del problema.** El PSO funciona mejor en problemas de altas dimensiones.
2. **Número de individuos.** Depende del problema en sí y la cantidad de dimensiones de este.
3. **Límite superior de p .** Depende del grado de oscilación en la trayectoria de las partículas buscado.
4. **Velocidad máxima $V_{\text{máx}}$.** Previene que las partículas abandonen rápidamente una región del espacio sin explorarla adecuadamente. Es usualmente proporcional al tamaño del problema.
5. **Tamaño de la vecindad.** Existe un compromiso entre la cantidad de vecinos (mayor área de búsqueda, menos posibilidad de caer en un mínimo local) y la rapidez de convergencia.
6. **Peso inercial.** Se puede mejorar la *performance* del PSO a través de su incorporación en el cálculo de velocidad:

$$\mathbf{v}_i(t) = \phi \mathbf{v}_i(t-1) + \text{componente } \textit{cognitiva} + \text{componente } \textit{social}.$$

dónde ϕ es el **peso inercial**. Este controla la influencia de las velocidades anteriores en las nuevas velocidades. Es conveniente que vaya decreciendo a través del tiempo, comenzando en un ϕ alto, para búsquedas en áreas más grandes, hasta un ϕ bajo, para refinar la búsqueda en áreas más chicas. Para que la convergencia sea posible, debe darse que: $\phi > \frac{1}{2}(c_1 + c_2) - 1$.

4.2.6. Comparación con los algoritmos genéticos

- Ambos usan reglas de transición **probabilísticas**.
- Ambos están basados en adaptación, pero en el PSO los cambios son obtenidos a través del aprendizaje entre iguales, no entre operadores de variación.
- El PSO tiene memoria: las partículas guardan registro de las mejores soluciones, y las velocidades anteriores son usadas para ajustar posiciones.
- El PSO no tiene función de *fitness*: el proceso de búsqueda es guiado por interacciones sociales entre partículas.

4.3. Colonias de Hormigas

4.3.1. Inspiración biológica

En las colonias de hormigas, la distribución y ejecución de las tareas está basada en diferencias anatómicas en los individuos (*que distinguen hormigas obreras de soldados, por ejemplo*) y en la **estigmergia**.

Se observan **conductas emergentes** cuando los agentes individuales prestan atención a sus vecinos inmediatos sin esperar órdenes, y actúan localmente. La acción colectiva de estos agentes produce el comportamiento global. Dentro de una colonia de hormigas, las mismas piensan localmente y actúan localmente, pero producen comportamiento global.

La **estigmergia** hace referencia al comportamiento distribuido dentro de la colonia de hormigas, siendo caracterizada por:

- La falta de una coordinación centralizada.
- La comunicación y la coordinación entre los individuos basada en modificaciones locales del ambiente.

- Retroalimentación positiva, que refuerza las acciones (*por ejemplo, las feromonas para seguir rastros de comida*).

La **estigmergía artificial** está definida como la comunicación indirecta mediada por modificaciones numéricas de los estados ambientales que son solos localmente accesibles por los agentes comunicadores. La esencia de modelar colonias de hormigas es el encontrar un modelo matemático que describa de forma precisa las características de **estigmergía** (*“invisible manager”*) correspondientes a cada individuo.

4.3.2. Feromonas

Las hormigas tienen la habilidad de siempre encontrar el **camino más corto** entre su nido y la fuente de alimento. Este comportamiento puede ser explicado por las feromonas arrojadas por cada hormiga. Durante la búsqueda por comida, y al regresar de la fuente de comida al nido, cada hormiga libera feromonas que se depositan en el camino. Para elegir un camino a seguir, las hormigas siguen el camino con mayor concentración de feromonas. El camino **más corto** va a tener un depósito de feromonas mucho más fuerte que el camino más largo, dado a que el **retorno** por ese camino desde la fuente de alimentos es **más rápido** (*nótese que liberan más feromonas de regreso al nido con alimento*). Además, el depósito de feromonas **se evapora con el tiempo**, por lo que la fuerza del depósito de feromonas se va decrementar mucho más rápido en el camino más largo que en el corto.

4.3.3. Algoritmo básico

1. Comportamiento inicial aleatorio.
2. Cuando encuentran una fuente de comida, se organizan y comienzan a seguir el nuevo camino.
 - 2.1 Mecanismo de reclutamiento: mayormente por **feromonas**, liberadas al regresar.
 - 2.2 Si otros encuentran el rastro, lo seguirán probablemente.
 - 2.3 El rastro se refuerza al ser seguido por más hormigas (pero se va evaporando).