UNIVERSIDAD NACIONAL DE INGENIERÍA
FACULTAD DE CIENCIAS
ESCUELA PROFESIONAL DE MATEMÁTICA

Redes Neuronales Artificiales. Arquitectura y Aplicaciones

Un proyecto presentado como Trabajo Práctico por:
Anthony Enrique Huertas Quispe

Supervisado por:
Edgard Kenny Venegas Palacios

2 de diciembre de 2016

Este trabajo esta dedicado
en especial a mi madre quien
confía en mis éxitos profesionales.

Índice general

Listings

Introducción

Desde el origen de la Informática, a causa de necesidades bélicas surgidas por la Segunda Mundial en los 40's, hasta los estudios y diseños actuales de la Inteligencia Artificial (IA), con el objetivo de desarrollar sistemas de aprendizaje en agentes abióticos con un mismo principio de funcionamiento llevado a cabo por los seres humanos, ha existido un campo de estudio sobre procesamientos neuronales que fundamentan tal avance técnológico que hoy en día permite que nuestras vidas tengan acceso a obtener resultados sencillos de problemas complejos, mediante un sistema de cálculo como el desarrollado por el cerebro humano, respecto al área en el que se presenten las dificultades. Dicho estudio sobre procesamientos neuronales en relación con la informática, conlleva a un tipo de desarrollo tecnológico denominado Computación Neuronal o **Redes Neuronales Artificales** (RNA) - *Artificial Neural Networks* -, el cual toma la arquitectura neurofisiológica del cerebro humano sumergiéndola hacia un sistema cibernético de estructura análoga tal que se adapte a distintos conceptos téoricos modernos referentes en diversos campos de aplicación. Si bien las técnologías desarrolladas por las RNA se han visto inicialmente fundamentadas en base a la teoría sobre el funcionamiento neuronal publicada en el artículo "A logical calculus of Ideas Imminent in Nervous Activity" (1943) de Warren McCulloch, un neurocientífico, y Walter Pitts, un logístico; se dice que las ideas básicas parten desde los siglos a.C, pues en dichas épocas ya se inventaban sistemas mecánicos que imitaban movimientos humanos y por ende se estaban desarrollando mecanismos de aprendizaje estructurando una capacidad o habilidad humana sobre máquinas, siendo ésta una tarea que procesa las RNA.
McCulloch y Pitts basaron sus ideas en estudios previos sobre el cerebro desde visiones tanto biológicas como el análisis sobre las ventajas de que se adopten los funcionamientos internos de ello en algún sistema no humano, y en su artículo se produndiza el entendimiento sobre cómo células básicas llamadas neuronas, que permanecen conectadas unas a otras, hacen que el cerebro produzca patrones altamente complejos en su afán de intercambiar información y desarrollar tareas, impulsando el nacimiento de la IA; además en dicho artículo, se presenta un modelo simplificado de una red neuronal mediante circuitos eléctricos denominada "neurona MCP', el cual ha contribuido de forma importante en el progreso de las RNA, originando el desarrollo de una red llamada *Perceptrón*, siendo éste el primer gran trabajo en computación neuronal llevada a cabo por Frank Rosenblatt (1957).

Cabe mencionar que las funciones de aprendizaje que forman parte de las redes neuronales, en su mayoría se encuentran fundamentadas, hasta la actualidad, en el trabajo de Donald Hebb mediante su libro "The Organization of Behaviour"(1949), en donde se establece la propuesta de una ley específica para la sinapsis de una neurona, con el objetivo de explicar el proceso de como se sistematiza el aprendizaje, mediante estudios que relacionan la psicología y la fisiología, formándose así la base de la *Teoría de las Redes Neuronales*.

El éxito técnologico desarrollado por Frank Rosenblatt (1957) sobre el diseño de la más antigua red neuronal "The Mark I perceptron", cuya función se encontraba caracterizada por el reconocimiento óptico de patrones, dio inicio a la *Primera Era de Oro de las Redes Neuronales* siendo Rosenblatt declarado fundador de la computación neuronal cuyos avances respecto al aprendizaje del perceptrón fueron detallados en su libro "Principles of Neurodynamics"(1959). Sin embargo, la primera red neuronal artificial con aplicación en la vida real fue el modelo *Adaline - Adaptative Linear Elements* - diseñada por Bernard Widrow (1960) en conjunto con sus estudiantes graduados, siendo Marcian Hoffel el más sobresaliente de ellos, quien hasta nuestras fechas es reconocido como el co-inventor del microprocesador. La era de oro finalizó a inicios de los años 60, a causa de un análisis crítico sobre las limitaciones de la red Perceptrón llevada a cabo por Marvin Minsky y Seymour Papert, cuyo trabajo se encuentra plasmado en el libro *Perceptrons* (1969), en donde se prueba matemáticamente que el Percetrón no presenta la capacidad de desarrollar problemas sencillos y el por qué todas las redes neuronales presentarían la misma desventaja, generando que las inversiones hacia las investigaciones de las RNA presenten un declive drástico. A pesar de ello, las investigaciones de las RNA continuaban en menor masa, tal es el caso de James Anderson quien desarrolló un modelo lineal llamado *Linear Associator*, basado en el Principio de Hebb: "El aprendizaje ocurre cuando ciertos cambios en una neurona son activados", y prontamente una extensión de tal modelo denominado *Brain State in a Box*.

A inicios de los años 80, resurgieron las investigaciones de las ARN, debido a las propuestas establecidas por varios científicos hacia el desarrollo de los neuro-ordenadores y las aplicaciones de redes neuronales, tal como es el caso de John Hopfield quien, mediante su artículo *Crossbar Associative Network* (1982) en conjunto con otros estudios, consiguió que cientos de científicos altamente calificados en todo el mundo se interesen en el campo de las RNA, dejándose atrás los casi 20 años de inactividad en tales estudios. La primera conferencia de las RNA en tiempos modernos se denominó "The IEEE International Conference on Neural Networks" (1987), establecida en San Diego, y prontamente fue formada la sociedad "International Neural Network Society - INNS"(1988) seguida por "Neural Computation" (1989) y "The IEEE Transactions on Neural Networks"(1990).

Actualmente, las investigaciones se desarrollan con sede en diversas universidades, como las situadas en California Institute of Technology, Massachussets Institute of Technology, University of California Berkeley y University of California San Diego.

Capítulo 1

Redes Neuronales Artificiales

Las **Redes Neuronales Artificiales (RNA)** son modelos matemáticos diseñados en base a un modelo sencillo denominado **Neurona Artificial**, (Artificial Neuron) que representa una estructura básica análoga a la de una neurona biológica con lo que respecta al mecanismo de procesamiento de información que ésta presenta. El fin común de toda red neuronal artificial es la de simular las conexiones biológicas entre neuronas, manteniéndose un funcionamiento que se adapte al entorno problemático en el que se desea incorporar, tomando estructuras de neuronas artificiales, cada una quizás con tareas distintas, y formando largas redes, donde tales conexiones definen una función de red. El logro obtenido de forma que se faciliten, aceleren y hagan más eficientes los distintos tipos de problemas que se intenten resolver, es gracias al diseño de una topología afinada sobre las RNA que establezcan una condición previa a sus usos; en detalle, significa la forma en la que se presenten sus interconecciones. Sin embargo, por el simple hecho de la diversidad de problemas, así como la capacidad biológica de resolverlas, la incoporación de técnicas de aprendizaje sobre el modelo antes de su uso es fundamental para hacer la correspondencia necesaria del modelo de RNA con el problema dado.

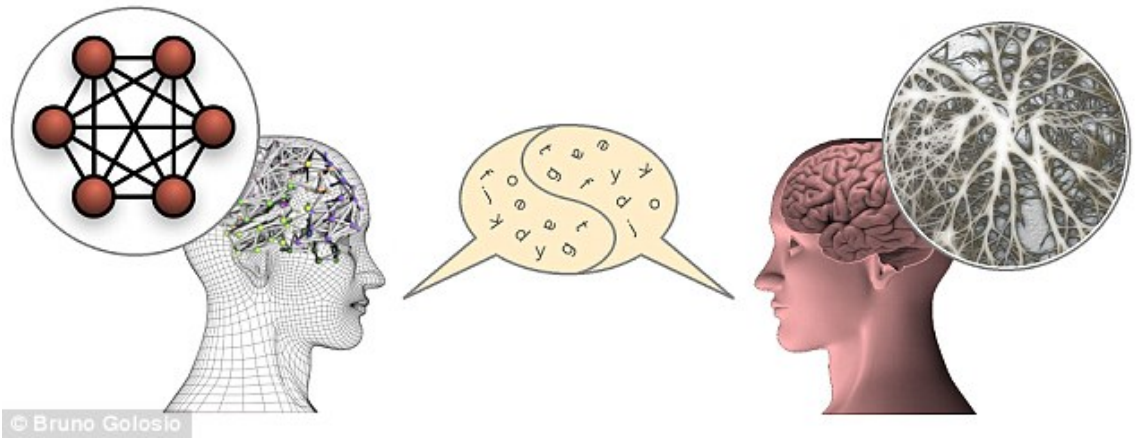


Figura 1.1: Redes Artificiales vs Redes Biológicas.

Estas técnicas de aprendizaje entrenan a una RNA mediante datos de entrenamiento que pueden consistir con datos de entrada y de salida (**aprendizaje supervisado**); con solo datos de entrada (**aprendizaje no supervisado**); o con datos de entrada adaptando señales de éxito y fallo (**aprendizaje por reforzamiento**), las cuales son técnicas de *Machine Learning*, siendo el campo de estudio que engloba el aprendizaje automatizado y por tanto las RNA son partes que complementan un estudio de dicho campo.
Las propiedades más atractivas del procesamiento de información llevada a cabo biológicamente son:

- El procesamiento de información es robusto y tolerante a fallos; debido a que, por varios años, no existe un deterioro en nuestras capacidades asociado a la pérdida diaria de miles y miles de neuronas.
- Los procesadores de información biológica son flexibles; indicando que su incorporación en nuevos entornos genera una adaptabilidad, es decir un aprendizaje sobre tal entorno, evitándose un tipo de reprogramación.
- Posee la capacidad de trabajar con datos confusos, probabilísticos, con ruido e inconsistentes; esto es a causa de la capacidad innata para manejar la incertidumbre, siendo esto computacionalmente posible gracias al desarrollo de una gran cantidad de programación sofisticada y cuando el contexto haya sido analizado en detalle.
- El sistema biológico que realiza estas funciones es altamente paralela, pequeña, compacta y disipa poca energía; siendo emulada por sistemas programables.

1.1. Neurona Artificial

Desde el punto de vista biológico, una neurona desarrolla el siguiente mecanismo: la información llega a causa de la sinapsis[1] ocurrida en las dendritas, para luego ser procesadas en el soma[2] dependiente del umbral de excitabilidad[3] de la neurona y finalmente enviadas por el axón hacia sus terminales produciéndose la sinapsis con otras neuronas. La neurona artificial fue diseñada de tal forma que siga el mismo mecanismo mediante una estructura computacional, en donde se establezcan instrucciones básicas de **multiplicación**, **suma** y **activación**; tal mecanismo es el siguiente: las informaciones se denominan entradas y llegan por vías de pesos, donde cada entrada es individualmente multiplicada por un peso, para posteriormente ser procesadas en el cuerpo artificial mediante una función de transferencia o activación

[1] Comunicación entre neuronas ocurridas en las terminales del axón de una y en las dendritas de otra, para enviar y recibir información respectivamente.
[2] Cuerpo neuronal.
[3] Es la condición para establecerse la actividad neuronal.

sobre la suma total de tales multiplicaciones previas y la sustracción de una cantidad denominada umbral, finalmente el resultado procesado se obtienen por salidas.

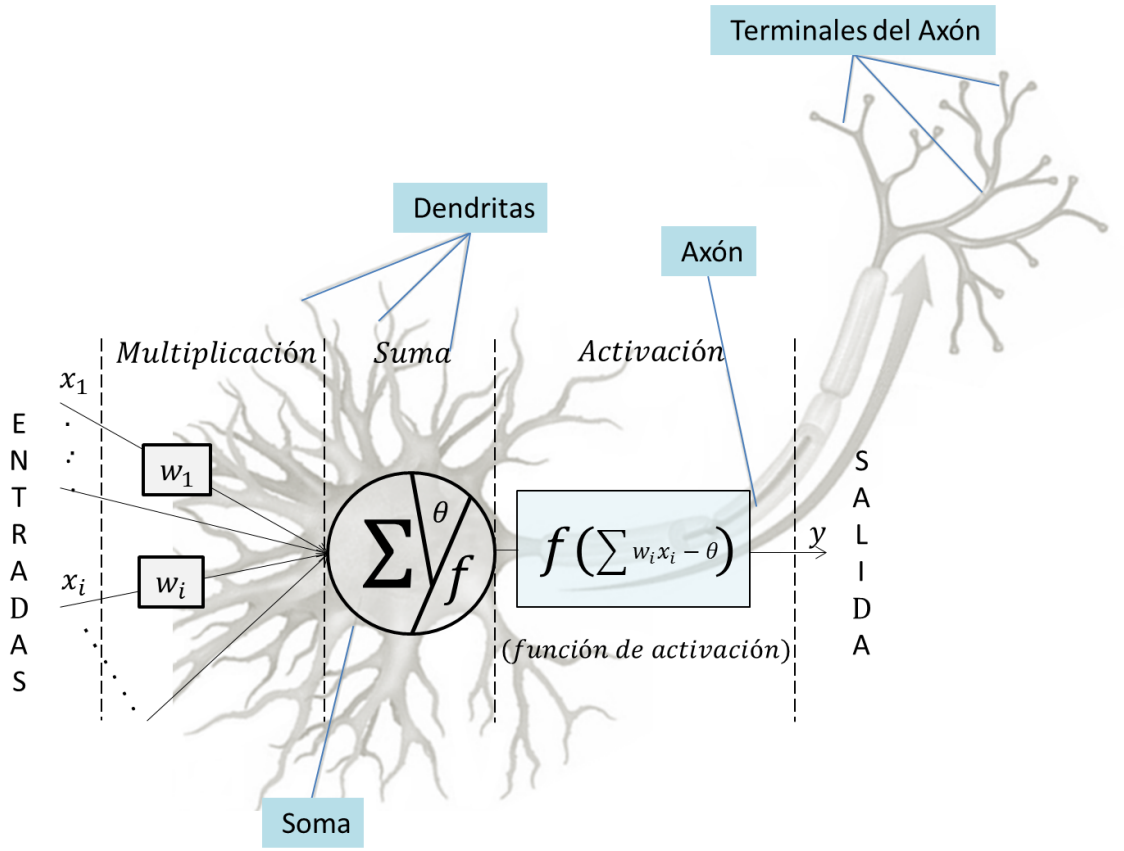


Figura 1.2: Red Neuronal Artificial (Simulación básica estructural).

Habiéndose ofrecido un alcance estructural de una neuronal artificial análoga a una neuronal biológica se obtienen las siguientes relaciones (Figura 1.2):

Neurona Biológica	**Neurona Artificial**
Informaciones	Entradas
Dendritas	Vía de Pesos
Soma	Cuerpo artificial
Umbral de excitabilidad	Umbral
Axón	Resultado procesado
Terminales del Axón	Salidas

Además de establecerse las siguiente fases operativas, para I valores de entrada:

- **Multiplicación**: Se realiza una multiplicación entre cada valor de entrada x_i y el peso w_i correspondiente a su vía de llegada hacia el cuerpo neuronal artificial:

$$w_i x_i. \tag{1.1}$$

- **Suma**: Se realiza la suma de los valores $w_i x_i$ obtenidos en la fase anterior conjuntamente con el valor negativo del umbral θ. Además, teniendo en cuenta que las sinapsis entre neuronas tienden a variar con el tiempo[4] (discreto), los valores de entrada y pesos vendrían a ser funciones implícitas de esta variable, resultando:

$$a = \sum_{i=1}^{I} w_i(t) x_i(t) - \theta. \tag{1.2}$$

  donde a es denominado *señal*.

- **Activación**: En esta fase, según el requerimiento del problema, una función denominada *función de activación* actúa sobre la señal a ejecutando un valor de salida dependiente del tiempo:

$$y(t) = f(a). \tag{1.3}$$

  donde y se denomina valor de salida.

Acotaciones:

- **Entradas**: Los valores de entrada $x_1, \ldots, x_I$ serán representados como un vector de entrada $\mathbf{x} = (x_1, \ldots, x_I)$.

- **Salidas**: El valor de salida "y" de una neurona artificial es único; y en caso esta neurona se conecte posteriormente con otras neuronas, entonces "y" correspondería ser un valor de entrada hacia cada una de ellas.

1.1.1. Funciones de Activación

Los principales tipos de funciones de activación llevadas a cabo por las neuronas artificiales se definen, acompañadas de su diagrama de representación, como sigue:

1. **Funciones escalonadas**: Son funciones que toman como conjunto de llegada dos posibles valores y son definidas como:

   a) **Función paso**: También denominada función limitador fuerte.

$$f(a) = \begin{cases} 1 & \text{si } \sum_{i}^{I} w_i x_i \geq \theta \\ 0 & \text{si } \sum_{i}^{I} w_i x_i < \theta \end{cases}$$

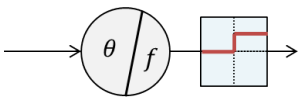


Figura 1.3: Función paso.

[4]Se hace referencia a cada iteración que son llevadas algorítmicamente, por tanto tiene que ser discreto.

b) **Función signo**: También denominada función limitador fuerte simétrico. La función suele denotarse como **Sgn()**.

$$Sgn(a) = \begin{cases} 1 & \text{si } \sum_i^I w_i x_i \geq \theta \\ -1 & \text{si } \sum_i^I w_i x_i < \theta \end{cases}$$

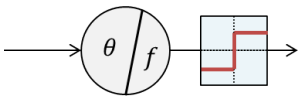


Figura 1.4: Función signo.

2. **Función lineal**: Hacemos referencia a funciones lineales y a ciertas transformaciones de carácter lineal sobre a.

a) **Función lineal**: En general la función Identidad es tomada como función lineal cuando se requiere su uso.

$$f(a) = \sum_i^I w_i x_i - \theta = a$$

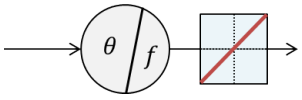


Figura 1.5: Función lineal.

b) **Función lineal positiva**:

$$f(a) = \begin{cases} a & \text{si } \sum_i^I w_i x_i \geq \theta \\ 0 & \text{si } \sum_i^I w_i x_i < \theta \end{cases}$$

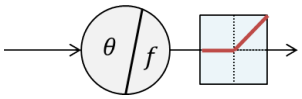


Figura 1.6: Función lineal positiva.

c) **Función lineal saturada**:

$$f(a) = \begin{cases} 1 & \text{si } a > 1 \\ a & \text{si } 0 \leq a \leq 1 \\ 0 & \text{si } a < 0 \end{cases}$$

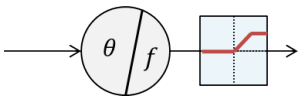


Figura 1.7: Lineal saturada.

d) **Función lineal saturada simétrica**:

$$f(a) = \begin{cases} 1 & \text{si } a > 1 \\ a & \text{si } -1 \leq a \leq 1 \\ -1 & \text{si } a < -1 \end{cases}$$

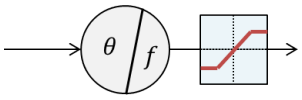


Figura 1.8: Función lineal saturada simétrica.

3. **Funciones no lineales**: Las funciones sigmoidales son las que mayormente se requieren como funciones no lineales.

   a) **Función sigmoidal logarítmica**:

$$f(a) = \frac{1}{1 + e^{-a}}$$

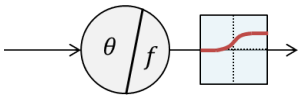


Figura 1.9: Función sigmoidal logarítmica.

   b) **Función tangencial sigmoidal hiperbólica**:

$$f(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$$

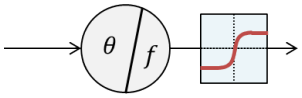


Figura 1.10: Función tangencial sigmoidal hiperbólica.

4. **Función Competitiva**:

$$f(a) = \begin{cases} 1 & \text{Neurona con mayor } a \\ 0 & \text{Resto de neuronas} \end{cases}$$

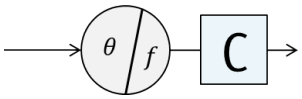


Figura 1.11: Función competitiva.

5. **Función Gaussiana**:

$$f(a) = Ae^{-Ba^2}$$

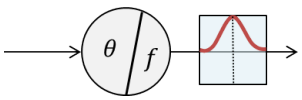


Figura 1.12: Función Gaussiana.

6. **Función Sinusoidal**:

$$f(a) = A\sin(\omega a + \varphi)$$

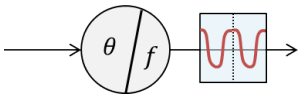


Figura 1.13: Función sinusoidal.

1.1.2. Neurona MCP

Warren McCulloch y Walter Pitts desarrollaron el primer modelo matemático de una neurona artificial en 1943, denominándola **Neurona MCP**, la cual tomaba a la función de paso como función de activación, y fueron quienes definieron los pesos como valores reales entre -1 y 1, cuyo tamaño correspondería a la cantidad influenciada para la obtención de la señal que se procese en el modelo, donde un peso positivo represente una sinapsis excitadora y un peso negativo, una sinapsis inhibidora.

$$\textbf{Pesos sinápticos excitadores:} \quad 0 \leq w \leq 1.$$
$$\textbf{Pesos sinápticos inhibidores:} \quad -1 \leq w < 0.$$

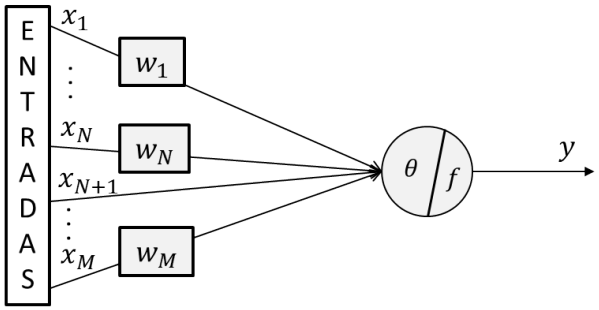


Figura 1.14: Diagrama de una neurona MCP. Los valores de entradas $x_1, \ldots, x_N$ corresponden a pesos sinápticos excitadores $w_1, \ldots, w_N$, y $x_{N+1}, \ldots, x_M$, a pesos sinápticos inhibdores $w_{N+1}, \ldots, w_M$. θ es el umbral de activación, f la función de activación y y el valor de salida.

El modelo, presenta las siguientes propiedades:

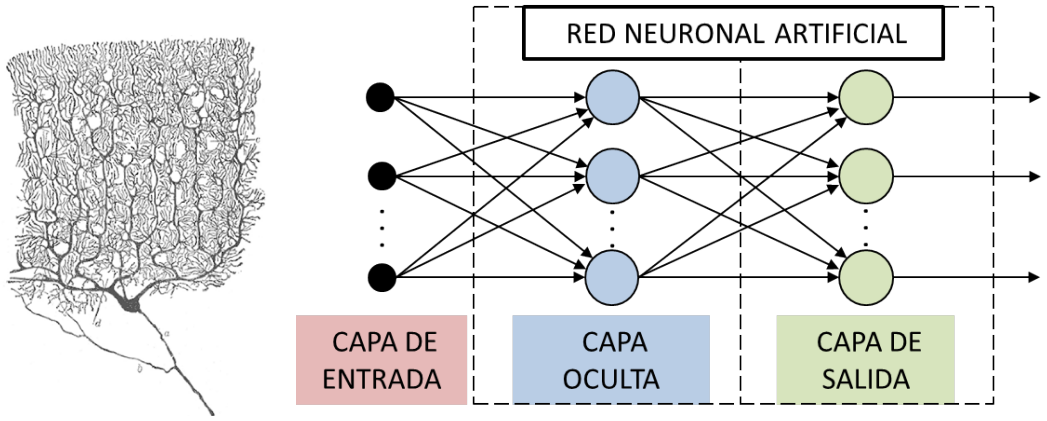
- La actividad neuronal es un proceso de "Todo o nada", indicando que una señal mayor o igual a cero implica actividad, en caso contrario, inactividad.
- Un número fijo de pesos sinápticos se excitan dentro de un período de adición latente con el fin de una actividad neuronal, independiente de la actividad previa de otra neurona y de su posición.
- El retraso sináptico es el único retraso significativo en el sistema nervioso, esto quiere decir que en dicho sistema existe una diferencia de tiempo entre la entrada de información y la información procesada.
- El fin de cualquier peso sináptico inhibidor es prevenir que una neurona ejecute un resultado.
- Estructura invariante en el tiempo.

1.2. Estructura de red

Una RNA presenta conexiones entre dos o más neuronas artificiales generándose la capacidad de resolver problemas complejos con aplicaciones en la vida real pues cada neurona corresponde a una tarea básica específica que en conjunto sistematiza un resultado propio de un problema inicial con alto grado de dificultad. Desde el punto de vista biológico, las sinapsis ocurridas entre neuronas ocurren según el posicionamiento que presentan en la corteza cerebral y sus conexiones. Este diseño natural colleva a uno artificial mediante una arquitectura, o topología, de redes neuronales artificiales de acuerdo al tipo de interconexiones que se requieran, definiéndose previamente las denominadas capas estructurales[5] (Figura 1.13.a):

- **Capa de entrada**: No forma parte de la red de neuronas, pues solo representa a la información inicial, es decir a los valores de entrada.
- **Capas ocultas**: Representan conjuntos de neuronas con conexiones a otras, recibiendo o emitiendo información. Sus elementos se denominan *unidades ocultas*.
- **Capa de salida**: Representa al conjunto de neuronas que generan las respuestas proporcionadas por la red, es decir los valores de salida. Sus elementos se denominan *unidades de salida*.

Se establecen dos tipos de arquitectura denominadas **RNA Alimentadoras** y **Recurrentes**; de las cuales la primera hace referencia a una de tipo acíclico, y la segunda a una de tipo cíclico.



(a) Dibujo de neuronas biológicas conectadas, desarrollado por Santiago Ramón y Cajal.

(b) Capas de una red neuronal artificial.

Figura 1.15: Redes neuronales biológicas vs Arquitectura de redes neuronales artificiales.

[5]Una RNA Simple posee solo capa de salida; una RNA Multicapa: (Bi-Nodal) solo posee una capa oculta, y (Tri-Nodal) o más capas poseen las tres capas estructurales.

1.3. Técnicas de Aprendizaje

En las RNA's, previo a su uso, se necesitan no solo establecerse topologías al estructurlas, las cuales son parte de un *modo operacional de activación*, sino también implementar técnicas (algoritmos de aprendizaje) que permitan la adaptabilidad de estos modelos a un entorno de problemas evitando reprogramarlas; es aquí cuando se dice que sobre el sistema actúa un *modo operacional de aprendizaje*.

1.3.1. Aprendizaje Supervisado

Este tipo de aprendizaje provee a una RNA la capacidad de predicción mediante la incorporación de datos previamente observados compuestos por un vector de entrada, denominado **input**, y un vector de salida, denominado **target**, denotados cada uno como:

$$(\underbrace{\overline{\mathbf{x}}_p}_{input}, \underbrace{\overline{\mathbf{z}}_p}_{target}) \tag{1.4}$$

siendo estos los datos de entrenamiento del modelo. Posteriormente, para determinar el buen aprendizaje en la red, se tendrá que evaluar inicialmente sobre los input y verificar que los vectores $\overline{\mathbf{y}}_p$ predichos por la RNA comparados con los targets $\overline{\mathbf{z}}_p$ presenten un error relativamente bajo; en caso contrario, los pesos sinápticos del modelo tendrían que ser ajustados hasta que que se cumpla ello.
Los pasos considerados en este tipo de aprendizaje son:

- *Primer paso*: Determinación del tipo de tarea mediante ejemplos.
- *Segundo paso*: Reunir los datos de entrenamiento que conjuntamente satisfagan la descripción del problema dado.
- *Tercer paso*: Descripción de los datos de entrenamiento con la finalidad de un tipo de comprensión para el diseño de un modelo de RNA.
- *Cuarto paso*: Establecer el aprendizaje y evaluar su funcionamiento mediante datos de prueba, consistente de datos reales no introducidos en el proceso.

Es común que la función a minimizar sea la del **Error Medio Cuadrático (Mean Squared Error, MSE)**, definido como:

$$E_k = \frac{1}{P}\sum_{p=1}^{P}\left(\overline{y}_{kp} - \overline{z}_{kp}\right)^2 \quad \forall k = 1, \ldots, K \tag{1.5}$$

siendo K la dimensionalidad de la capa de salida. Además de minimizar el error suele requerirse el algoritmo denominado Propagación hacia atrás (**Back-Propagation**) cuando el aprendizaje es implementado sobre redes Binodales a más.

1.3.1.1. Back-Propagation

El algoritmo Back-Propagation es comúnmente usado en RNA's multicapa de tipo alimentadoras (sección 2.2) aplicando la regla delta[6] generalizada y permitiendo la propagación de errores de una capa hacia la capa que la precede en el sentido del flujo de las sinapsis. Se establecen los siguiente pasos computacionales:

1. **Forward Pass** (Propagación de activación): Tomando una red binodal, se da la siguiente propagación del flujo de información:

   - Capa 1: Si J unidades en la capa reciben información $x \in \mathbb{R}^I$, se ejecutan las siguientes salidas:

   $$y_j^{(1)} = \sum_{i=1}^{I} w_{ji}^{(1)} x_i, \quad \forall j = 1, \ldots, J. \tag{1.6}$$

   - Capa 2: Si K unidades en la capa reciben información de las J unidades de la capa 1, se ejecutan las siguientes salidas:

   $$y_k^{(2)} = \sum_{j=1}^{J} w_{kj}^{(2)} y_j^{(1)}, \quad \forall k = 1, \ldots, K. \tag{1.7}$$

2. **Backward Pass** (Propagación de los errores hacia atrás): El proceso Forward Pass se mantiene sobre cualquier capa. Ahora, suponiendo que la capa 1 es la última capa oculta y la capa 2 es la capa de salida entonces los errores vendrían dados como:

   - Capa de salida (Capa 2):

   $$e_k = \overline{z}_{kp} - \overline{y}_{kp}, \quad \forall k = 1, \ldots, K. \tag{1.8}$$

   - Capa oculta (Capa 1):

   $$e_j = \sum_{k=1}^{K} w_{kj}^{(2)} e_k, \quad \forall j = 1, \ldots, J. \tag{1.9}$$

El método generaliza la ecuación 1.8 de presentarse más capas ocultas que precedan a la capa de salida, y se diseñan las actualizaciones de pesos siguientes:

$$w_{ji} = w_{ji} + \eta e_j x_i = w_{ji} + \Delta w_{ji}. \tag{1.10}$$

donde η es denominada taza de aprendizaje.

[6] Sea x_i el valor que ingresa de la unidad j a la unidad de salida t, a través del peso w_{tj}. La regla delta establece una ecuación de actualización de pesos de la siguiente forma: $\Delta w_{tj} = \eta(z_t - y_t)x_j$.

1.3.1.2. RProp

El método flexible de propagación hacia atrás (*Resilient Back-Propagation*, **RPROP**) fue diseñado con el objetivo de generar un mecanismo de actualización de pesos sinápticos más rápido y eficiente que el método usual de Back-Propagation sobre problemas con mayor complejidad. El algoritmo RProp elimina la influencia perjudicial provocada por el tamaño de la derivada parcial en la actualización de los pesos. y viene dada, siendo t la actual iteración, como:

$$w_i(t+1) \quad = \quad w_i(t) - sign\left(\frac{\partial E(t)}{\partial w_i}\right)\Delta^i(t). \tag{1.11}$$

Tomando como base el proceso de adaptación *signo-dependencia*, se determinan los nuevos valores actualizados $\Delta_i(t)$ denominado peso *valor-actualizado* dado como sigue:

$$\Delta_i(t) = \begin{cases} \eta^+.\Delta_i(t-1), & \text{si } \dfrac{\partial E(t-1)}{\partial w_i}.\dfrac{\partial E(t)}{\partial w_i} > 0 \\ \eta^-.\Delta_i(t-1), & \text{si } \dfrac{\partial E}{\partial w_i}(t-1).\dfrac{\partial E(t)}{\partial w_i} < 0 \\ \Delta_i(t-1), & \text{Otros casos.} \end{cases} \tag{1.12}$$

donde $\eta^+ > 1$ y $\eta^- \in [0,1]$.

Algoritmo 1: RPROP

para w_i **hacer**
  si $\dfrac{\partial E}{\partial w_i}(t-1).\dfrac{\partial E(t)}{\partial w_i} > 0$ **entonces**
    $\Delta_i(t) = \text{mín}\left(\Delta_i(t-1)\eta^+, \Delta_{\text{máx}}\right)$;
    $w_i(t+1) = w_i(t) - sign\left(\dfrac{\partial E(t)}{\partial w_i}\right)\Delta^i(t)$;
  en otro caso
    si $\dfrac{\partial E}{\partial w_i}(t-1).\dfrac{\partial E(t)}{\partial w_i} < 0$ **entonces**
      $\Delta_i(t) = \text{máx}\left(\Delta_i(t-1)\eta^-, \Delta_{\text{mín}}\right)$;
      si $E(t) > E(t-1)$ **entonces**
        $w_i(t+1) = w_i(t-1)$
      fin si
    en otro caso
      $w_i(t+1) = w_i(t) - sign\left(\dfrac{\partial E(t)}{\partial w_i}\right)\Delta^i(t)$;
    fin si
  fin si
fin para

1.3.2. Aprendizaje no Supervisado

Este tipo de aprendizaje provee a una RNA la capacidad de aprender sobre datos sin supervisión explícita, es decir, con datos de entrenamiento que no presenten targets. Por lo tanto, una RNA que incorpore este aprendizaje determina como los datos se autoorganizan a causa del flujo de actividad llevada a cabo desde el ingreso de los valores de entrada hasta la ejecución de los valores de salida en el interior de la red. La técnica más común es la de *Clustering* donde se trata de agrupar datos en distintos grupos de acuerdo a ciertas similitudes encontradas. Los paradigmas con mayor uso son el aprendizaje Hebbiano y el aprendizaje Competitivo.

1.3.2.1. Aprendizaje Hebbiano

Donald Hebb conjeturó lo siguiente:

"*Cuando el axón de una neurona A está suficientemente cerca de excitar una neurona B y repetidamente o persistentemente toma parte de su activación, algún proceso de crecimiento o cambio metabólico toma lugar en una o ambas neuronas tal que la eficiencia de A, como una de las neuronas que activa B, se incrementa.*"

Considerando este hecho, Hebb determinó la siguiente regla de aprendizaje (*Hebbiano*) para la actualización de los pesos sinápticos, que se diferencia de la dada en (1.10):

$$\Delta w_{ji} = \eta x_i y_j = \eta x_i \sum_i w_{ji} x_i \tag{1.13}$$

1.3.2.2. Aprendizaje Competitivo

Este tipo de aprendizaje tiene como objetivo categorizar los datos en grupos tomando en cuenta a aquellas neuronas en un red que logren activarse al recibir flujos de información, refiriéndose a éstas como ganadoras en una competencia entre todas las demás neuronas. Lo seguro es que ocurra un proceso de autoorganización sobre los datos en base a la estructura implícita que mantienen. Un aprendizaje de competencia simple conduce a la creación de células madre, quienes son las responsables en la activación posterior de neuronas.
El mecanismo de competencia simple consiste en la siguiente regla de actualización de pesos:

$$\Delta w_{ji} = \eta(x_i - w_{ji}) \tag{1.14}$$

para una neurona ganadora j, con el objetivo de que vuelva a lograrlo en un futuro cuando se presenten valores de entrada similares.
El problema que puede presentarse en este tipo de aprendizaje es que podría gene-

rarse una neurona dominante en el proceso dejando a otras neuronas inactivas todo el tiempo.

1.3.2.3. Reducción de Dimensionalidad

Las técnicas de reducción de dimensionalidad consisten en reducir las altas dimensionalidades de los datos sin pérdida significativa de información, con el objetivo de que una modelo RNA genere un menor esfuerzo computacional. El aprendizaje Hebbiano, en combinación con términos de decadencia de pesos[7] puede ser visto como una reducción de dimensionalidad similar a la técnica PCA.
Dos de las técnicas más comunes son las siguientes:

- **PCA**: El análisis de componentes principales (Principal component analysis) consiste en hallar los eigenvalores, y sus correspondientes eigenvectores ortogonales, de la matriz de covarianza generada por los datos, pues dado que estos explican la variabilidad existente en los datos, solo bastaría en mantener una dimensión menor con aquellos eigenvalores y eigenvectores (componentes principales) que representen suficiente variabilidad.
- **ICA**: El análisis de componentes independientes (Independent component analysis) mantiene un enfoque similar al PCA pero sin restricción a la ortogonalidad de eigenvectores. Esta técnica busca determinar un conjunto de vectores independientes dentro de un conjunto de datos que no se limitan a correlaciones lineales.

1.3.3. Aprendizaje por Reforzamiento

Una RNA diseñada con este aprendizaje permite que no se requiera una incorporación inicial de datos sino mas bien estos son generados por interacciones con el entorno. El aprendizaje por reforzamiento logra que una RNA tome ciertas acciones con el objetivo de maximizar una recompensa a largo plazo, es decir que obtenga una característica que permita desarrollar tareas de igual forma como se realizó o mejor.

[7]La decadencia de pesos hace referencia a que estos mantengan una tendencia hacia el valor cero mediante el método $w_{ji}^{actual} = (1 - \varepsilon)w_{ji}^{pasado}$ solo cuando se tiene un número alto de pesos en la neurona correspondiente; el objetivo es evitar el peligro de sobreajuste y por ende una mala mejora generada por los datos de prueba.

Capítulo 2

Selección de Variables de Entrada (IVS)

Una RNA mantiene un funcionamiento de naturaleza lineal o no lineal, donde en este último caso, sumado a la complejidad inherente al modelo, trae consigo dificultades sobre la **Selección de las Variables de Entrada (Input Variable Selection, IVS)** que entrenen exitosamente una arquitectura de red neuronal, las cuales se presentan debido a un número de variables disponibles con valor muy alto o correlaciones existentes entre posible variables de entrada, generándose redundancias, o variables que tienen poco o ningún poder predictivo.
Esta tarea IVS suele ser llevada a cabo durante la fase de aprendizaje, identificándose aquellas variables redundantes o con ruido con el objetivo que la RNA sea entrenada solo con las variables de entrada más sobresalientes denominadas *candidatas*.
El problema IVS toma interés en el siguiente modelo de una RNA que proporciona una función de activación no lineal

$$Y = f(X) + \epsilon \tag{2.1}$$

donde Y es la variable de interés, X es la variable que denota vectores de entrada I-dimensionales cuyas componentes son denotadas por X_i, y ϵ es el ruido[1].
Las consideraciones relevantes que genera IVS sobre el rendimiento del modelo RNA son las siguientes:

- **Relevancia**: El rendimiento de una RNA puede verse afectado en caso se seleccionen muy pocas variables o si el conjunto seleccionado no sea lo suficientemente informativo. En este caso, se tendrían que reconsiderar los datos disponibles y optar por una mejor selección.
- **Esfuerzo Computacional**: Un mayor número de variables de entrada aumenta la carga de cálculo y por ende un velocidad de entrenamiento menor.

[1]Se denomina "ruido" al error entre un valor de salida real para una vector de entrada y la variable de interés generada por el modelo para tal vector

- **Dificultad de entrenamiento**: El entrenamiento de una RNA se vuelve mucho más difícil debido a la inclusión de variables de entrada redundantes e irrelevantes pues muchas más iteraciones se vuelven necesarias en el algoritmo de entrenamiento con el objetivo de determinar un error óptimo global, y no local, que se suma a la carga computacional del modelo.

- **Dimensionalidad**: El incremento de observaciones sobre datos que inicien el entrenamiento de un modelo de RNA generan que estos a su vez incrementen el número de sus características y por tanto su dimensionalidad. Algunas arquitecturas de RNA son muy sensibles a este aumento; sin embargo, otras poseen la capacidad de eludirla ignorando las variables irrelevantes o redundantes.

- **Comprensibilidad**: El modelamiento de una RNA se encuentra más centrado en el descubrimiento de conocimiento de datos (knowledge discovery from data, KDD) y la minería de datos; en donde, la primera tiene como objetivo el entrenamiento basado en la observación del proceso para luego interrogar a la red logrando una mejor comprensión del proceso que ha aprendido.

Los algoritmos IVS se presentan en tres clases principales, siendo estos los algoritmos *contenedores*, de *inserción* y de *filtro*. Sin embargo, estos a su vez necesitan de ciertos criterios de optimización, la cual se detallará en la siguiente sección, como de estrategias de búsqueda: (i) *selección hacia adelante*: Es computacionalmente eficiente pues no considera todas las posibles combinaciones de las variables de entrada y solo opta por un subconjunto pequeño de ellas, determinándose posiblemente un óptimo local; (ii) *eliminación hacia atrás*: Es un método con proceso inverso al de selección hacia adelante e ineficiente en comparación con éste pues requiere mejorar y evaluar sobre múltiples modelos complejos de RNA's; (iii) *búsqueda exhaustiva*: Evalúa sobre todas las posibles combinaciones de las variables de entrada aproximando un óptimo global, siendo factible el proceso cuando la dimensionalidad es baja e infactible en caso contrario,; (iv) *búsqueda heurística*: Es usado cuando el espacio de búsqueda es de dimensionalidad alta y aproxima un óptimo global; (v) *selección escalonada*: Es una extensión de la selección hacia adelante.

2.1. Criterio de Optimización

El rendimiento predictivo definen los criterios de optimización para el entrenamiento de una RNA, que son las pruebas necesarias para determinar la influencia de las variables de entrada seleccionadas. La medida adoptada para tal rendimiento predictivo es comúnmente E, siendo el MSE dado en (1.5).
La estrategia corresponde en escoger el modelo que establezca el mínimo valor E; sin embargo, esto no corresponde necesariamente a un modelo óptimo pues un número grande de variables de entrada tienden a ser sesgadas como resultado de un sobreajuste. Esto se regula usando el criterio de Mallows'C_ρ, definiendo una medida del

umbral relativo y la varianza de un modelo con ρ variables como

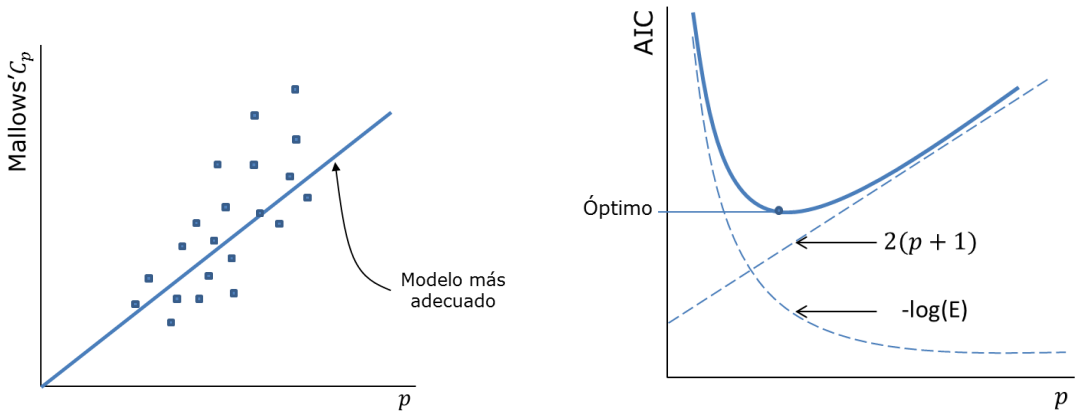
$$C_\rho = \frac{E}{\sigma_D^2} - N + 2\rho \tag{2.2}$$

donde σ_D^2 es el residual para un modelo entrenado completo usando las D posibles variables de entrada.
Además tanto el criterio de Mallows como el criterio de información Akaike definido como

$$AIC = -N \log E - 2(\rho + 1), \tag{2.3}$$

determinan el número óptimo de variables de entrada penalizando el sobreajuste. Con respecto a AIC, el modelo óptimo resulta de minimizar esta función con respecto al número de parámetros ρ. El criterio de optimización mantiene diferencias especiales respecto a cada uno de los algoritmos principales:

- Algoritmos Contenedores: La optimización hace uso exclusivo de la medida (2.2), y de las ecuaciones que penalizan el sobreajuste.
- Algoritmos de Inserción: Se considera una regularización de pesos para minimizar la complejidad manteniendo el rendimiento predictivo.
- Algoritmos de Filtro: Se establece un análisis estadístico de las variables candidatas de acuerdo a lo siguiente: (i) Máxima relevancia, (ii) mínima redundancia y (iii) mínima redundancia-máxima relevancia.



(a) Mallows'C_p. (b) AIC.

Figura 2.1: Comparación entre el criterio C_ρ y AIC.

2.2. Algoritmos Contenedores

Estos tipos de algoritmos son los de formulación más simple y son definidos por la elección de la arquitectura del modelo. La eficiencia se mide por la habilidad del modelo para representar relaciones entre los datos y por como los modelos de pruebas son construidos y evaluados.

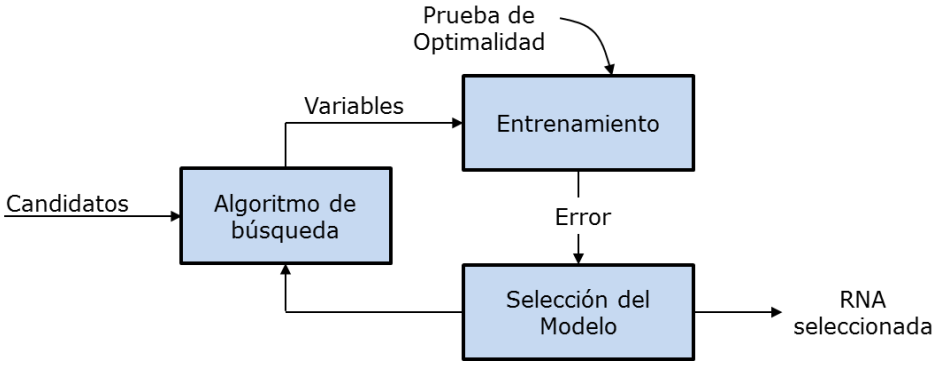


Figura 2.2: Algoritmos Contenedores (Diagrama).

2.2.1. Estrategias de búsqueda

A continuación se presentarán las estrategias correspondientes a los algoritmos contenedores que son técnicas de *subconjuntos anidados*, siendo denominado de tal forma pues se explora sobre regiones que comprenden conjuntos superpuestos por variables que están siendo adicionadas constantemente.

- *Selección hacia adelante*: Se establece un entrenamiento iterativo inicialmente con las D variables de entrada y se selecciona aquella que maximice la eficiencia basado en el criterio de optimización; el proceso continúa con $D-1$. En cada iteración se adiciona una variable candidata a la variable de entrada seleccionada previamente. La selección se concluye al adicionar una variable que reduzca la eficiencia del modelo.

- *Eliminación hacia atrás*: Se toman inicialmente las D variables de entrada y se eliminan las menos importantes una por una. Esta estrategia consiste en remover una variable de entrada y examinar el funcionamiento del modelo sin ella y así sucesivamente.

- *Búsqueda exhaustiva*: Selecciona el mejor conjunto de variables de entrada que establezcan una mejor eficiencia basado en el criterio de optimización. Siendo D el número total de variables de entrada se logran evalúar 2^D conjuntos, siendo este proceso factible cuando la dimensionalidad del conjunto de variables candidatas es baja.

- *Búsqueda heurística*: Esta técnica es usada cuando el espacio de búsqueda es de alta dimensionalidad, y establece un mecanismo de enfoque de búsqueda aleatoria sobre regiones que traen consigo buenos resultados. Esta búsqueda retiene cierto grado de aleatoriedad que no garantiza una solución óptima pero si eficiente.

- *Selección escalonada*: Los modelos lineales son construidos agregando iterativamente una variable de entrada al modelo, y reestimando sus coeficientes. El proceso continúa hasta que se satisfaga algún criterio de optimización.

2.2.2. Regresión Variable Simple

La regresión variable simple (SVR) es contruida de tal forma que use cada variable candidata y las categorice según la optimalidad que genere de acuerdo al criterio de optimización. El algoritmo de determinación de entrada para una red neuronal de regresión generalizada, GRNN (sección 3.2.3), denominada **GRIDA** es un ejemplo de SVR para categorizar las variables de entrada.

Algoritmo 2: GRIDA

Datos: X (conjunto de variables de entrada).
para $x \in X$ **hacer**
  Entrenar un GRNN;
  Determinar E_x;
  para $b = 1, \ldots, 100$ **hacer**
    $\epsilon \leftarrow x$;
    Estimar $E_{\epsilon,b}$;
  fin para
  Estimar $E_\epsilon^{(95)}$;/* 95^{avo} `percentil` */
  si $E_x > E_\epsilon^{(95)}$ *o* $E_x > \Theta$;/* Θ `es algún valor límite` */
  entonces
    Remover x de X.
  fin si
fin para
devolver X

Considerando cada variable a su vez, un GRNN es entrenada y luego el error medio cuadrático es determinado para un conjunto de datos de prueba. Una variable es rechazada si el error del modelo excede a una cota de confianza $E_\epsilon^{(95)}$ o límite Θ establecido.

2.3. Algoritmos de Insersión

Un algoritmo de insersión establece tanto la selección de las variables de entrada como su evaluación dentro del algoritmo de entrenamiento de una RNA. El diseño de estos algoritmos permite un proceso iterativo de actualización de pesos considerando de manera individual el impacto de cada variable en el rendimiento del modelo.

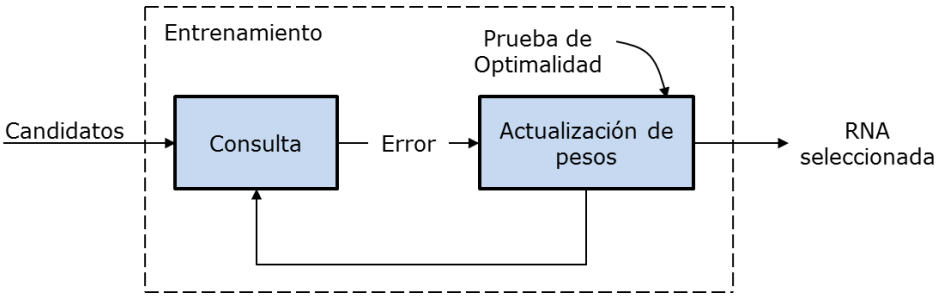


Figura 2.3: Algoritmos de insersión (Diagrama).

2.3.1. Estrategias de búsqueda

- *Selección escalonada*: La técnica usada es la de *regresión escalonada*, la cual permite la mejora de modelos de regresión lineal. El proceso es análogo que en los algoritmos contenedores.
- *Eliminación hacia atrás*: La técnica desarrollada para algoritmos de insersión es la de *eliminación de características recursivas*, la cual envuelve un proceso iterativo de entrenamiento, usando inicialmente todos las variables candidatas y removiendo una o más de ellas en cada iteración basado en la magnitud de los pesos sinápticos correspondiente a cada variable de entrada.

2.4. Algoritmos de Filtro

El diseño de los algoritmos de filtro se definen por una medida de relevancia que permite distinguir las variables de entrada importantes, y los criterios de optimización. Esta medida de relevancia necesita evaluar cada relación entre las variables candidatas y las variables de salida. Las clases principales son las basadas en correlaciones lineales y las basadas en medidas de información teórica.

2.4.1. Estrategias de búsqueda

- *Selección hacia adelante*: La variable candidata más relevante es seleccionada primero, luego la selección procede iterativamente identificando la siguiente variable candidata más relevante, hasta que se satisfaga el cirterio de optimización.

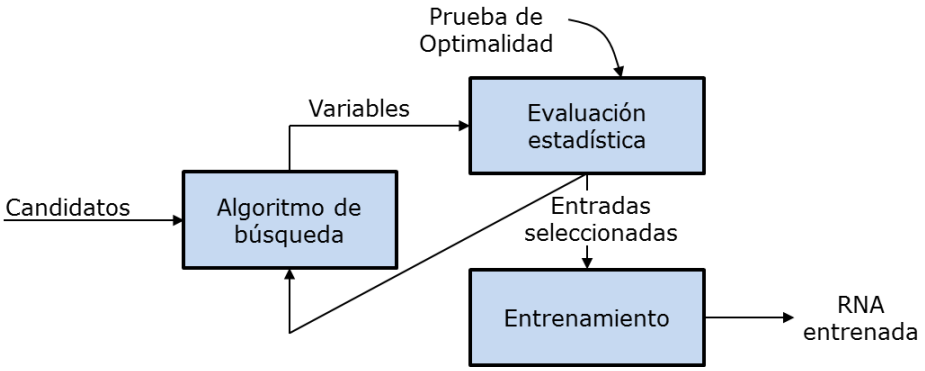


Figura 2.4: Algoritmos de filtro (diagrama).

- *Eliminación hacia atrás*: Los candidatos menos relevantes son removidos iterativamente hasta que se satisfaga el criterio de optimización.

2.4.2. Correlación

- *Correlación lineal*: Para la estadística multivariada, la correlación de Pearson es la de mayor uso, y es definida como:

$$R_{XY} = \frac{Var(X,Y)}{\sqrt{Var(X)Var(Y)}} \tag{2.4}$$

  donde $Var(X,Y)$ es la covarianza entre las variable X (de entrada) y Y (de salida).

- *Correlación parcial*: En el caso que muchas variables candidatas proporcionen la misma información con respecto a la variable de salida, se establece una correlación parcial $R_{XY|Z}$ que mide la correlación entre X e Y dejando de lado la relación presente entre Y e Z, y es definida como:

$$R_{XY|Z} = \frac{R_{XY} - R_{XZ}R_{YZ}}{\sqrt{\left(1 - R_{XZ}^2\right)\left(1 - R_{YZ}^2\right)}} \tag{2.5}$$

- *Autocorrelación*: Este tipo de correlación se usa para la determinación de patrones repetitivos en un análisis de series de tiempo, y es definido como:

$$R_k = \frac{\sum_{i=1}^{n-k}(x_i - \overline{x})(x_{i+k} - \overline{x})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{2.6}$$

  donde k se denomina decrecimiento y $\overline{x}$ valor promedio respecto a X.

La mejora de una RNA puede no ser óptima usando el análisis de correlación pues el IVS es basado en modelos lineales.

2.4.3. Teoría de la Información (Entropía)

Las limitaciones generadas por el análisis de correlación, promueve el interés en medidas que identifiquen y cuantifiquen la dependencia de un modelo no lineal. La teoría de la información (o teoría matemática de la comunicación), propuesta por C. Shannon y W. Weaver, se ocupa de medir la información, su representación y procesamiento. El concepto de entropía, o *entropía de Shannon*, es el valor esperado de la información contenida en cada modelamiento.

2.4.3.1. Información mutua (IM)

La información mutua (IM) es una medida de dependencia basada en la entropía de Shannon, y determinada por la siguiente ecuación:

$$I(X;Y) = \int\int p(x,y) \log \frac{p(x,y)}{p(x)p(y)} dxdy \tag{2.7}$$

donde p representa una función de probabilidad. MI provee una aproximación al error asociado con la estimación de la esperanza $E[y|X]$ determinada usando el *Teorema de Bayes*[2], donde el mínimo error conlleva a maximizar $I(X,Y)$. Además, la relación que IM mantiene con respecto a la correlación lineal (2.5) es:

$$I(X;Y) = \frac{1}{2} \log(1 - R^2_{XY}) \tag{2.8}$$

- *IM-Selección de características*: Este algoritmo aborda las deficiencias en base a la correlación lineal, usando un grado de redundancia, β, cuyo incremento influencia en la redundancia de selección de valores candidatas.

Algoritmo 3: IM-Selección de características

Datos: $X \to \phi$;
mientras $|X| < k$ **hacer**
  para $c \in C$ **hacer**
    $I(c, Y|X) \leftarrow I(c;Y) - \beta \sum_{x \in X} I(c;x)$;
  fin para
  $c_s \leftarrow \text{máx}_c\{I(c, Y|X)\}$;
  Mover c_s a X;
fin mientras
devolver X;

[2]El teorema de Bayes establece una regla de probabilidad condicional: $p(y|x) = \frac{p(y)p(x|y)}{p(x)}$.

- *IM parcial*: Este algoritmo es de estructura similar a *IM-Selección de características* solo que se basa en un criterio de estimación directa de la distribución del error.

Algoritmo 4: IM parcial

Datos: $X \to \phi$;
mientras $C \neq k$ **hacer**
  Construir estimador de regresión kernel $\hat{m}_Y(X)$;
  $u = Y - \hat{m}_Y(X)$;
  para $c \in C$ **hacer**
    Construir estimador de regresión kernel $\hat{m}_c(X)$;
    $v = c - \hat{m}_c(X)$;
    Estimar $I(v; u)$;
  fin para
  $v_s \leftarrow \text{máx}_v\{I(v, u)\}$;
  para $b = 1, \ldots, B$ **hacer**
    Escoger aleatoriamente v_s^*;
    Estimar $I_b \leftarrow I(v_s^*; u)$;
  fin para
  Encontrar límite de confianza $I_b^{(95)}$;
  si $I(v_s, u) > I_b^{(95)}$ **entonces**
    Mover c_s a X;
  fin si
  Break;
fin mientras
devolver X

Capítulo 3

Arquitectura de RNA's y Aplicaciones

En el primer capítulo se realizó la comparación entre una red neuronal biológica y una artificial junto al proceso de activación que pudiese ser llevado a cabo por una unidad neuronal. En este capítulo se darán a conocer los procesos llevados a cabo por las arquitecturas de redes neuronales más importantes desarrolladas en los últimos tiempos junto a algunas aplicaciones de la vida real, siendo éstas entrenadas junto a criterios de optimización, implícitos a los procesos, y reglas de aprendizaje vistos en los capítulos anteriores. Ciertamente los diseños son construidos con el objetivo de situarse en un entorno problemático específico y confrontarla dependiendo de las características propias de dicho entorno; sin embargo, se entandarizará el proceso teniendo en cuenta que estas arquitecturas se ven divididas en dos tipos de acuerdo a la presencia de fases cíclicas (**redes alimentadoras**) o acíclicas(**redes retroalimentadoras**) en su sistema de flujo de activación.
Es de importante mención, que si bien una RNA es entrenada también lo puede ser una única neurona artificial, es por ello que se establecerá en la sección siguiente un entrenamiento básico neuronal de un Perceptrón simple.
Cabe mencionar que las aplicaciones desarrolladas en este capítulo han sido desarrolladas usando los lenguajes de programación PYTHON, R y MATLAB, habiéndose tomado como prioridad al primero de estos. Además, se detallará el proceso desarrollado en tales aplicaciones como las librerías más importantes usadas para el diseño de los codigos, es decir las que vienen directamente relacionadas con el tipo de RNA que se trabaje; pues es fundamental que el lector mantenga un conocimiento sobre ciertas librerías básicas (como numpy, matplotlib, random, etc).
Las notaciones para los datos de entrenamiento son de la forma $(\overline{\mathbf{x}}_p, \overline{\mathbf{z}}_p)$ o $(\overline{\mathbf{x}}(t), \overline{\mathbf{z}}(t))$, siendo t la representación de las iteraciones, según sea el caso en un aprendizaje supervisado, sino simplemente $\overline{\mathbf{x}}_p$.

3.1. Perceptrón Simple (PS)

El Perceptrón simple (**PS**) no es una red sino una unidad neuronal. El PS fue el primer diseño de neurona artificial con aplicación real desarrollado por Frank Rosenblatt, tomando como base el modelo matemático de la neurona MCP, la cual hace uso de la función signo, y sistematizando un aprendizaje de **error-corrección**.

PROCESO:
Sean $(\overline{\mathbf{x}}(t), \overline{z}(t))$ los P datos de entrenamiento, donde $\overline{\mathbf{x}} = \{\overline{x}_i\}_{i=1}^{I}$, $\overline{z} \in \mathbb{R}$.

1. **Límite de activación**: Denotando $\overline{x}_0 = -1$ y $w_0 = \theta$, las restricciones de la función signo, vendrían como:

$$\sum_{i=1}^{I} w_i \overline{x}_i \gtreqless \theta \Rightarrow \sum_{i=1}^{I} w_i \overline{x}_i \gtreqless 0. \tag{3.1}$$

2. **Regla de aprendizaje**: La determinación de los pesos para una siguiente iteración t, haciendo uso de una tasa de aprendizaje $\eta(t)$ en cada una de ellas, viene expresada de la siguiente forma:

$$w_i(t+1) \;=\; w_i(t) + \underbrace{\eta(t)[\overline{z}(t) - \overline{y}(t)]\overline{x}_i(t)}_{\Delta w_i(t)},\ i \in \{0, \ldots, I\}. \tag{3.2}$$

   Se denomina *regla de adaptación de incremento fijo* si η es constante.

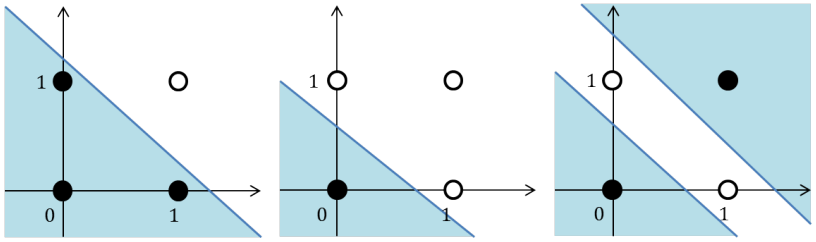
3. **Error y corrección**: Los errores $\overline{z}(t) - \overline{y}(t)$ determinan tres posibles cambios para la corrección de los pesos en cada iteración t:

$$w_i(t+1) \;=\; \begin{cases} w_i(t) & \text{si } \overline{z}(t) = \overline{y}(t). \\ w_i(t) + 2\eta(t)\overline{x}_i(t) & \text{si } \overline{y}(t) = -1 \text{ y } \overline{z}(t) = 1. \\ w_i(t) - 2\eta(t)\overline{x}_i(t) & \text{si } \overline{y}(t) = 1 \text{ y } \overline{z}(t) = -1. \end{cases} \tag{3.3}$$

LIMITACIONES:
El diseño de un perceptrón simple no satisface clasificaciones correctas cuando los datos no son linealmente separables, es decir que el PS separa los datos solo en dos clases modelando la siguiente ecuación de recta e hiperplanos H_1, H_2:

$$L : w^T\mathbf{x} = \theta,\ H_1 : \{\mathbf{x} :\ w^T\mathbf{x} \geq \theta\},\ H_2 : \{\mathbf{x} :\ w^T\mathbf{x} < \theta\}. \tag{3.4}$$

Algunos ejemplos más comunes de funciones de separación lineal que un PS logra aprender a desarrollar son las funciones AND y OR, a diferencia de una función de separación no lineal como XOR que el PS no es capaz de aprender (ver Figura 3.1).



(a) Función AND. (b) Función OR. (c) Función XOR.

Figura 3.1: Las funciones (a) AND y (b) OR pueden ser aprendidas por el PS, a diferencia de la función XOR.

Algoritmo 5: Percetrón Simple

Datos: $(\overline{\mathbf{x}}(t), \overline{z}(t),\ t = 1, \ldots, P;\ w_i(1)$ (aleatorios para $i = 0, \ldots, I)$.

contador $= 0$;
mientras *correctos* $< P$ *ó contador*< 100 **hacer**
  $x_0(t) = -1$;
  correctos $= 0$:
  para $t = 1, \ldots, P$ **hacer**
    $\overline{y}(t) = signo\left(\sum_{i=0}^{I} w_i(t)x_i(t)\right)$;
    si $\overline{z}(t) == \overline{y}(t)$ **entonces**
      correctos++;
    en otro caso
      para $i = 0, \ldots, I$ **hacer**
        $w_i(t+1) = w_i(t) + \eta(t)[\overline{z}(t) - \overline{y}(t)]\overline{x}_i(t)$;
      fin para
    fin si
  fin para
fin mientras
devolver $w_i(t)$ *para* $i = 0, \ldots, I$.

3.1.1. Aplicación (Separación lineal)

Se entrenará un perceptrón simple con el objetivo de generar una recta (3.4) que separe datos previamente clasificados, con inputs bidimensionales y targets unidimensionales.

En el código Listing 3.1, el código es diseñado en PYTHON haciendo uso de la librería sklearn.linear_model en donde se importará la clase **perceptron**. Los datos de entrenamiento son almacenados en cadenas para luego ser clasificados en hiperplanos generados por la definición de un recta producto de un modelo lineal generado por la función **perceptrón()**, verificándose luego la eficiencia de la clasificación.

Listing 3.1: PS/Entrenamiento (Fuente: Stamford Research, Lenguaje Python).

```
# LIBRERIAS IMPORTADAS
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import perceptron
%matplotlib inline
# DATOS
d = np.array([[2, 1, 2, 5, 7, 2, 3, 6, 1, 2, 5, 4, 6, 5],
              [2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 7]
              ])
# TARGETS
t = np.array([0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1])
# PLOTEO DE LOS DATOS
colormap = np.array(['r', 'k'])
plt.scatter(d[0], d[1], c=colormap[t], s=40)
d90 = np.rot90(d)
d90 = np.rot90(d90)
d90 = np.rot90(d90)
# CREAR EL MODELO
net = perceptron.Perceptron(n_iter=100, eta0=0.002)
net.fit(d90,t)
# IMPRIMIR RESULTADOS
print "Prediccion " + str(net.predict(d90))
print "Real      " + str(t)
print "Eficiencia   " + str(net.score(d90, t)*100) + "%"
# PLOTEO DE LA SEPARACION LINEAL
plt.scatter(d[0], d[1], c=colormap[t], s=40)
print "Coefficient 0 " + str(net.coef_[0,0])
print "Coefficient 1 " + str(net.coef_[0,1])
print "Bias " + str(net.intercept_)
ymin, ymax = plt.ylim()
w = net.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(ymin, ymax)
yy = a * xx - (net.intercept_[0]) / w[1]
plt.plot(yy,xx, 'k-')
```

Output:

Prediccion [00011001001111]
Real [00011001001111]
Eficiencia 100.0 %

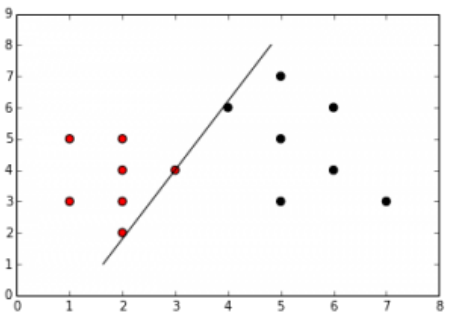


Figura 3.2: Datos de entrenamiento clasificados por hiperplanos.

Finalmente, habiéndose sistematizado el aprendizaje en el perceptrón, se imprimen resultados predichos para nuevos valores de entrada usando Listing 3.2.

Listing 3.2: PS/Evaluación (Fuente: Stamford Research, Lenguaje Python).

```
# CREANDO NUEVOS DATOS
nX = np.random.random_integers(10, size=(2,50))
print nX
nX90 = np.rot90(nX)
nX90 = np.rot90(nX90)
nX90 = np.rot90(nX90)
# PREDICCION
prediction = net.predict(nX90)
print prediction
# PLOTEO
plt.scatter(nX[0],nX[1], c=colormap[prediction],s=40)
ymin, ymax = plt.ylim()
w = net.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(ymin, ymax)
yy = a * xx - (net.intercept_[0]) / w[1]
plt.plot(yy,xx, 'k-')
```

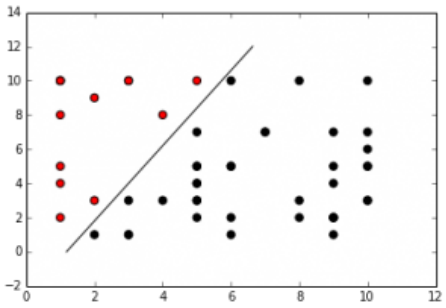


Figura 3.3: Datos de evaluación clasificados por el modelo.

FUNCIÓN perceptron(n_iter=5, eta0=1.0):
Está función corresponde a la clase **sklearn.linear_model.perceptron()**, la cual es de muy buen uso en aprendizajes de larga escala mantiendo el proceso señalado en Algoritmo 5. Sus parámetros son: **n_iter**: El número de iteraciones; **eta0**: Tasa de crecimiento. Los métodos de la función de uso en la aplicación son:

- **fit(X,y)**: Ajusta el modelo lineal con los datos de entrenamiento **(X,y)**.
- **predict(X)**: Predice el vector de salida para una muestra **X**.
- **score(X,y)**: Retorna la eficiencia media.

Observación: La función **rot90()** es usada en el código, haciendo referencia a una rotación de 90° a causa de un arreglo visual en los resultados.

3.2. RNA's Alimentadoras

Una arquitectura de red neuronal artificial alimentadora (Feed-Forward Artificial Neural Network, **FNN**) presenta una topología de tipo acíclica, es decir que al activarse una neurona en la red, y ejecutar un resultado propia de ésta, no vuelve a activarse pero si las neuronas que la proceden en el sentido de interconexión. Esto se reduce planteándose la siguiente condición:

"*Las informaciones fluyen y se complementan unas con otras en una sola dirección, sin retrocesos*".

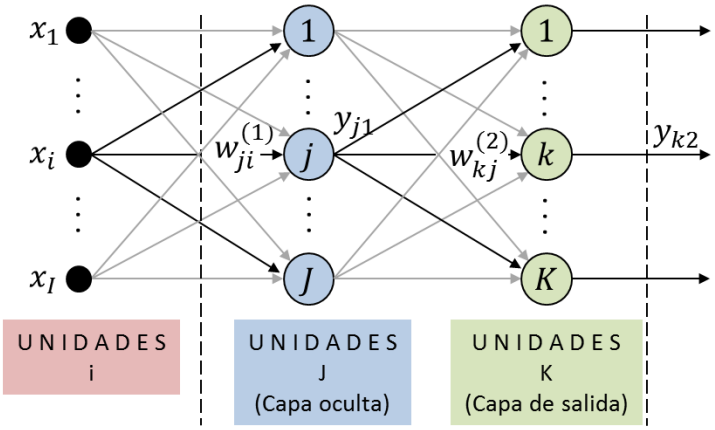


Figura 3.4: Arquitectura de una RNA Alimentadora.

Descripción analítica de una FNN binodal (Figura 1.5):

- **Capa de entrada**:

$$x_1, x_2, \ldots, x_I.$$

- **Capa oculta**:

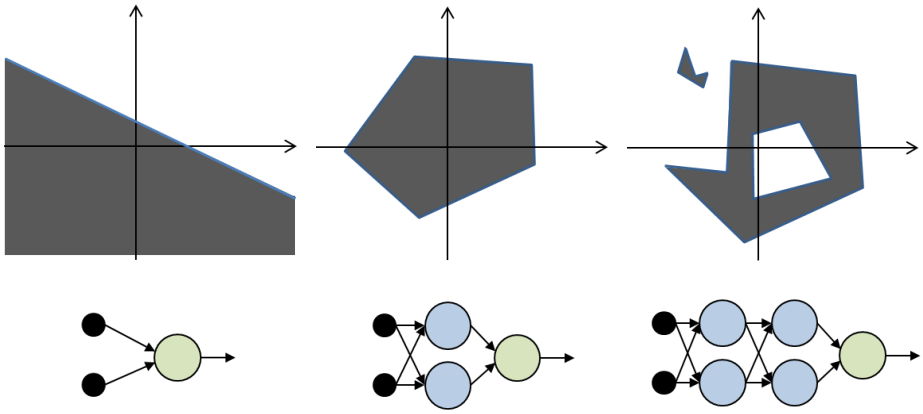
$$y_j^{(1)} = f_{j1}\left(\sum_{i=1}^{I} w_{ji}^{(1)} x_i - \theta_j^{(1)}\right).$$

- **Capa de salida**:

$$y_k^{(2)} = f_{k2}\left(\sum_{j=1}^{J} w_{kj}^{(2)} y_j^{(1)} - \theta_k^{(2)}\right) = f_{k2}\left(\sum_{j=1}^{J} w_{kj}^{(2)} f_{j1}\left(\sum_{i=1}^{I} w_{ji}^{(1)} x_i - \theta_j^{(1)}\right) - \theta_k^{(2)}\right)$$

Una arquitectura FNN puede ser altamente amplia, sin embargo, las descripciones matemáticas son análogas a las descritas anteriormente y por tanto correspondería el desarrollo de un sistema especializado que optimice el desarrollo de tal proceso.

3.2.1. Perceptrón Multicapa (MLP)



(a) Perceptrón Simple. (b) Perceptrón binodal. (c) Perceptrón trinodal.

Figura 3.5: (a)Perceptrón Simple vs (b)(c) Perceptrón Multicapa .

Los Perceptrones Multicapa (Multi-Layer Perceptron, **MLP**) inicialmente fueron diseñadas para lidiar con las limitaciones que presentaban los PS al realizar algunos problemas técnicamente sencillos, como la separación de datos mediante una función XOR (Figura 3.1.c), y su denominación es por la presencia de capas ocultas en su arquitectura a diferencia de un PS. Los MLP, con el uso de funciones de paso, contruyen cotas de decisión lineal (Figura 3.5); además para obtenerse un mayor alcance, se usan otras funciones como la función sigmoidal logarítmica.

PROCESO:

1. **Flujo de activación**: La salida de la capa de entrada para una MLP con una sola capa oculta (Single-Hidden-Layer MLP) es expresada como:

$$G(\mathbf{x}) = f\left(W^T\mathbf{x} - \theta\right) = f\left(W^T\mathbf{x} + \mathbf{b}\right) \tag{3.5}$$

donde $G : \mathbb{R}^I \to \mathbb{R}^K$, I es la dimensión del vector entrada, K la dimensión del vector salida, f la función de activación, W la matriz de pesos y $\theta = -\mathbf{b}$ el vector de umbrales.
Sea $\mathbf{y}^{(c)} = \left(\mathbf{y}_1^{(c)}; \ldots; \mathbf{y}_{J_c}^{(c)}\right)$ el vector de salida de la c-ésima capa, la cual posee J_c unidades, y $\mathbf{y}^{(c-1)}$ su entrada correspondiente. $\mathbf{y}^{(0)} = \mathbf{x}$ el vector de entrada a la red, $\mathbf{y}^{(C)} = \mathbf{y}$ vector de salida de la red. La salida de la capa c es:

$$\mathbf{y}^{(c)} = f_c\left(\left(W^{(c)}\right)^T \mathbf{y}^{(c-1)} + \mathbf{b}^{(c)}\right) = f(\mathbf{a}^{(c)}) \tag{3.6}$$

donde la salida correspondiente a cada unidad oculta j de la capa c es:

$$y_j^{(c)} = f_c\left(\left(W_j^{(c)}\right)^T \mathbf{y}^{(c-1)} + b_j^{(c)}\right) = f_c\left(a_j^{(c)}\right) \tag{3.7}$$

2. **Función de activación**: La función sigmoidal logística es una función de activación convientemente diferenciable dada por:

$$f_c(\mathbf{x}) = \frac{e^{W^{(c)}\mathbf{x}+\mathbf{b}^{(c)}}}{\sum_d e^{W^{(d)}\mathbf{x}+\mathbf{b}^{(d)}}}. \tag{3.8}$$

3. **Función costo**: Sean los P datos de entrenamiento $(\overline{\mathbf{x}}_p, \overline{\mathbf{z}}_p)$, y los obtenidos por la red de la forma $(\overline{\mathbf{x}}_p, \overline{\mathbf{y}}_p)$, entonces los pesos son ajustados minimizando la siguiente función costo:

$$E_{CE} = \sum_{p=1}^{N}\sum_{k=1}^{K}\left[\overline{z}_{kp}\log\left(\overline{y}_{kp}\right) + (1-\overline{z}_{kp})\log\left(1-\overline{y}_{kp}\right)\right] \tag{3.9}$$

donde E_{CE} es la *entropía cruzada* (error medido a causa del Backpropagation).

4. **Pesos Sinápticos**: Usando descenso de gradiente se tiene que:

$$\Delta w_{ji}^{(c)} = -\eta\frac{\partial E\left(w_{ji}^{(c)}\right)}{\partial w_{ji}^{(c)}} \tag{3.10}$$

donde η es la tasa de aprendizaje. Usando la regla de la cadena para derivadas, y vista para una Single-Hidden Layer MLP, tenemos que para $d = 1, 2$:

$$\frac{\partial E_{CE}\left(w_{ji}^{(c)}\right)}{\partial w_{hl}^{(d)}} = -\sum_{p=1}^{N}\sum_{k}^{K}\left[\frac{\overline{z}_{kp}-\overline{y}_{kp}}{\overline{y}_{kp}\left(1-\overline{y}_{kp}\right)}\right]\frac{\partial \overline{y}_{kp}}{\partial w_{hl}^{(d)}} \tag{3.11}$$

Algoritmo 6: Perceptrón Multicapa

Datos: $(\overline{\mathbf{x}}_p, \overline{\mathbf{z}}_p)\ \forall p \in \{1,\ldots,N\}$; C; $W^{(c)}\ \forall c \in \{1,\ldots,C\}$; δ.

Establecer conexiones entre cada capa c hacia la capa previa $c-1$.
`/* Generar pesos aleatorias por cada capa */`
Calcular $E(w_{jk}^{(c)})$;
`/* Tasa de aprendizaje */`
Seleccionar η;
mientras $E_{CE} < \delta$ **hacer**
  para $c = 1,\ldots,C$ **hacer**
    $\Delta w_{hl}^{(c)} = -\eta\dfrac{\partial E(w_{ij}^{(c)})}{\partial w_{hl}^{(c)}}$;
  fin para
fin mientras

3.2.1.1. Aplicación (Función lógica XOR)

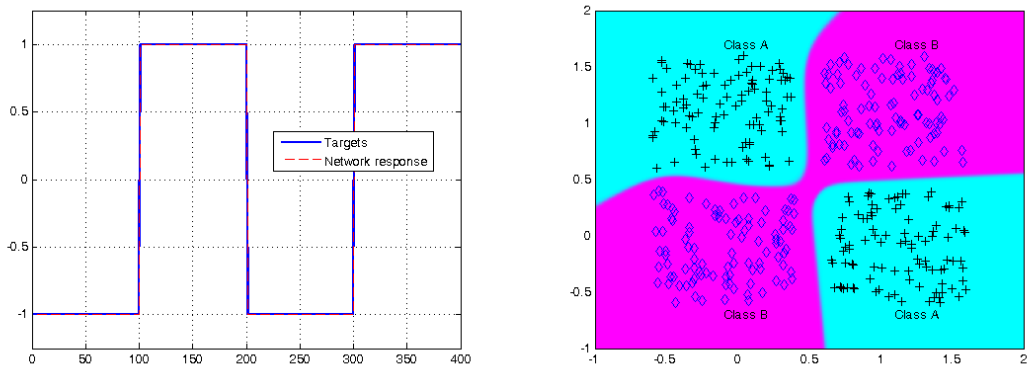
El primer problema que el PS no podía lidiar era el de incorporar una aprendizaje tal como la función XOR; es por ello que a continuación veremos como la arquitectura MLP logra definir agrupamientos de forma que adopte tal aprendizaje.

En el código Listing 3.3, los datos de entrenamiento son clasificados en 4 grupos y 2 clases diferentes, a su vez distribuidos en un eje de coordenadas de forma que no puedan ser divididos mediante una recta. Se diseña un MLP con 2 capas ocultas (una de 5 unidades y la otra de 3 unidades), para posteriormente entrenar a la red con los datos de entrenamiento. Se obtienen los resultados predichos por la red para los datos de entrenamiento y se comparan con sus targets para observar el buen funcionamiento (Figura 3.6 (a)). Se diseña una malla usando la red entrenada para determinarse los espacios de clasificación (Figura 3.6 (b)).

Listing 3.3: MLP/Funcion **XOR** (Fuente: 2012 Primoz Potocnik, Lenguaje: Matlab).

```
close all, clear all, clc, format compact
% Numero de muestras de cada clase.
K = 100;
% Definir 4 agrupamientos de los datos de entrada.
q = .6; % offset of classes
A = [rand(1,K)-q; rand(1,K)+q];
B = [rand(1,K)+q; rand(1,K)+q];
C = [rand(1,K)+q; rand(1,K)-q];
D = [rand(1,K)-q; rand(1,K)-q];
% Visualizacion de los agrupamientos.
figure(1)
plot(A(1,:),A(2,:),'k+')
hold on
grid on
plot(B(1,:),B(2,:),'bd')
plot(C(1,:),C(2,:),'k+')
plot(D(1,:),D(2,:),'bd')
text(.5-q,.5+2*q,'Class A')
text(.5+q,.5+2*q,'Class B')
text(.5+q,.5-2*q,'Class A')
text(.5-q,.5-2*q,'Class B')
%a y c son de una clase; b y d, de otra clase.
a = -1; %  a  |  b
c = -1; % -------
b =  1; %  d  |  c
d =  1; %
% Definir entradas
P = [A B C D];
% definir las salidas reales
T = [repmat(a,1,length(A)) repmat(b,1,length(B)) ...
     repmat(c,1,length(C)) repmat(d,1,length(D)) ];
% Crear la RNA
net = feedforwardnet([5 3]);
% Entrenamiento de la red.
net.divideParam.trainRatio = 1; % conjunto de entrenamiento
net.divideParam.valRatio   = 0; % conjunto de validacion
net.divideParam.testRatio  = 0; % conjunto de prueba
% Entrenamiento de la RNA
[net,tr,Y,E] = train(net,P,T);
```

```
40 % Mostrar la RNA
41 view(net)
42 figure(2)
43 plot(T','linewidth',2)
44 hold on
45 plot(Y','r--')
46 grid on
47 legend('Targets','Network response','location','best')
48 ylim([-1.25 1.25])
49 % Generar una malla
50 span = -1:.005:2;
51 [P1,P2] = meshgrid(span,span);
52 pp = [P1(:) P2(:)]';
53 % Simulacion de la RNA en una malla.
54 aa = net(pp);
55 % translate output into [-1,1]
56 %aa = -1 + 2*(aa>0);
57 % Visualizacion de las regiones de clasificacion.
58 figure(1)
59 mesh(P1,P2,reshape(aa,length(span),length(span))-5);
60 colormap cool
61 view(2)
```



(a) Comparación entre los Targets y la respuesta de la red (Network response)

(b) Agrupamiento y espacios de clasificación por la red.

Figura 3.6: Visualización del problema XOR resuelto con MLP.

FUNCIÓN feedforwardnet(size):
La función corresponde al lenguaje de programación MATLAB. Diseña la red MLP, usando el siguiente parámetro: **size**, es una lista cuya longitud es la cantidad de capas ocultas y cuyos elementos son las unidades correspondientes a cada una de ellas. Se hace uso de los siguientes métodos:

- **train(red,P,T)**: Entrena la **red** diseñada por la función mediante backpropagation, teniendo como parámetros a las agrupaciones **P** de los vectores de entrada de los de entrenamiento con su conjunto de targets **T**.

Observación: En línea 35,36,37 se dividen los datos, siendo el 100 % tomados como datos de entrenamiento y 0 % para los datos tanto de prueba como de evaluación.

3.2.2. RNA de Base Radial (RBF)

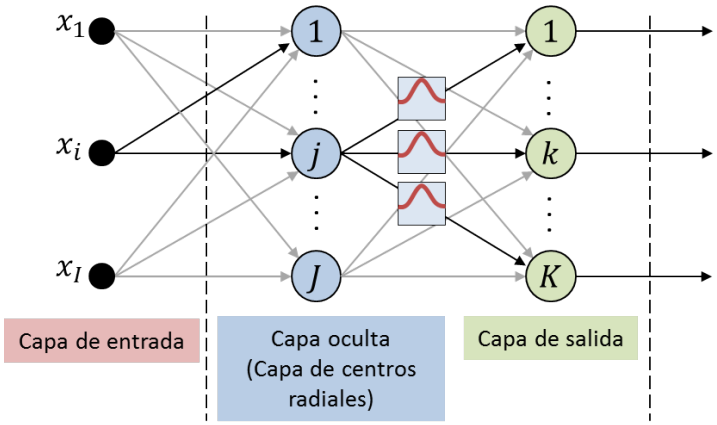


Figura 3.7: Red Neuronal Artificial de Base Radial.

La arquitectura de una red neuronal de base radial (Radial Basis Function, **RBF**) consta de una capa oculta denominada *Capa de centros radiales* pues desarrolla una funcion de activación radial, comúnmente la función gaussiana:

$$\phi_j(\mathbf{x}) = \phi\left(\|\mathbf{x} - \mathbf{s}_j\|\right) = \exp\left\{-\frac{\|\mathbf{x} - \mathbf{s}_j\|^2}{2\sigma_j^2}\right\} \tag{3.12}$$

donde $\mathbf{s}_j$ es el vector central y σ_j un valor paramétrico, correspondientes a la neurona j en la capa de centros radiales.

PROCESO:

- **Mapeo**: Sean P datos de entrenamiento $(\overline{\mathbf{x}}_p, \overline{\mathbf{z}}_p)$. Los vectores $\overline{\mathbf{z}}$ generalmente son generados por funciones abyacentes $g_k(\overline{\mathbf{x}})$ con cierto ruido. El objetivo es aproximar g_k con $\overline{y}_k$ de la forma

$$y_k(\mathbf{x}) \;=\; \sum_{j=0}^{J} w_{kj}\phi_j(\mathbf{x}) \;\; \forall k \in \{1, \ldots, K\}. \tag{3.13}$$

- **Selección aleatoria de centros fijos**: La selección de centros radiales puede ser resuelta usando el *Algoritmo 5/K-means clustering*[1] minimizando:

$$\widetilde{J} \;=\; \sum_{j=1}^{J}\sum_{p\in S_j} \|\mathbf{x}^p - \mathbf{s}_j\| \tag{3.14}$$

  donde S_j son subconjuntos disjuntos asignados a ciertos vectores $\mathbf{x}_p$.

[1]Este algoritmo toma K subconjuntos (clusters) disjuntos de particiones de puntos, determinándose K centros por cada cluster.

Definido (3.12), donde $\{\mathbf{s}_j\} \subset \{\overline{\mathbf{x}}_p\}$, los σ_j se relacionan o con la distancia máxima o promedio entre los centros $\mathbf{s}_j$ escogidos:

$$\sigma_j = \frac{d_{\text{máx}}}{\sqrt{2J}} \quad , \quad \sigma_j = 2d_{promedio} \tag{3.15}$$

- **Errores (Capa de salida)**: Definimos la sumas cuadrática de errores en la capa de salida como:

$$E = \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(\overline{z}_{kp} - \overline{y}_{kp}\right)^2 \tag{3.16}$$

El problema de minimización a resolver analíticamente es el siguiente:

$$\frac{\partial E}{\partial w_{kj}} = \sum_{p}\left(\overline{z}_{kp} - \sum_{l=0}^{J} w_{kl}\phi_l(\overline{\mathbf{x}}_p)\right)\phi_j(\overline{\mathbf{x}}_p) = 0 \tag{3.17}$$

- **Pesos de salida**: Las ecuaciones para los pesos son escritos en una matriz, definiendo las siguientes matrices con componentes $W = (w_{kj})$, $\Phi = (\phi_j(\overline{\mathbf{x}}_p))$ y $Z = (\overline{z}_{kp})$. Obteniendo:

$$\Phi^T\left(Z - \Phi W^T\right) = 0 \;\Rightarrow\; W^T = \underbrace{\left(\Phi^T\Phi\right)^{-1}\Phi^T}_{\Phi^\dagger} Z \tag{3.18}$$

Algoritmo 7: K-means Clustering (para centros radiales)

Datos: $(\overline{\mathbf{x}}_p, \overline{\mathbf{z}}_p)\ \ \forall p \in \{1,\ldots,P\},\ S_j \subset \{1,\ldots,p\}\ \ \forall j \in \{1,\ldots,J\}$.

1 $S_1,\ldots,S_J = \underset{S_j(j=1,\ldots,J)}{\text{mín}} \left\{ \widetilde{J} = \sum_{j=1}^{K}\sum_{p\in S_j}\left\|\overline{\mathbf{x}}_p - \frac{1}{n(S_j)}\sum_{p\in S_j}\overline{\mathbf{x}}_p\right\| \right\}$;

2 **para** $j = 1,\ldots,J$ **hacer**

3 $\quad \mathbf{s}_j = \frac{1}{n(S_j)}\sum_{p\in S_j}\overline{\mathbf{x}}_p$;

4 **fin para**

5 **devolver** $\{\mathbf{s}_j\}_{j=1}^{J}$

3.2.2.1. Aplicación (Interpolación)

La red RBF puede ser usada para interpolar datos n-dimensionales. En listing 3.4, se generan datos tridimensionales en donde una dimensión representa los targets. Se establece una sección de prueba $[-2,2]\times[-2,2]$ usando **meshgrid()** en donde

se verán interpolado los datos de dicha sección usando el modelo generado por los datos de entrenamiento usando **Rbf()**.

Listing 3.4: RBF/Interpolación (Fuente: SciPy Cookbook, Lenguaje: Python).

```
import numpy as np
from scipy.interpolate import Rbf
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from matplotlib import cm
# Datos de entrenamiento y area de prueba
x = np.random.rand(100)*4.0-2.0
y = np.random.rand(100)*4.0-2.0
z = x*np.exp(-x**2-y**2)
ti = np.linspace(-2.0, 2.0, 100)
XI, YI = np.meshgrid(ti, ti)
# Usar RBF
rbf = Rbf(x, y, z, epsilon=2)
ZI = rbf(XI, YI)
# Ploteo de resultados
n = plt.normalize(-2., 2.)
plt.subplot(1, 1, 1)
plt.pcolor(XI, YI, ZI, cmap=cm.jet)
plt.scatter(x, y, 100, z, cmap=cm.jet)
plt.title('RBF interpolation - multiquadrics')
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.colorbar()
plt.savefig('rbf2d.png')
```

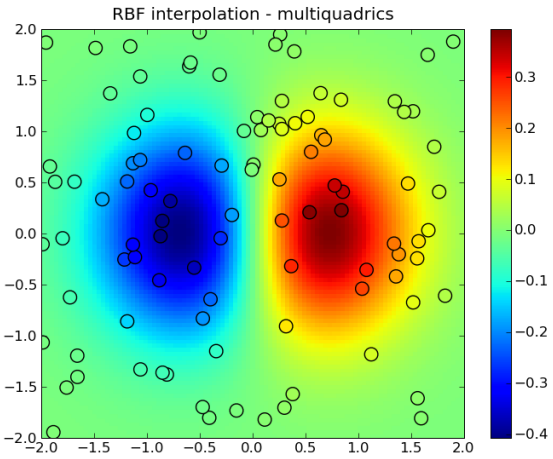


Figura 3.8: Mapa de calor para interpolación de datos.

FUNCIÓN Rbf(arg, función, epsilon)**:
La librería **scipy.interpolate** incluye la función **Rbf**, la cual posee los siguientes parámetros: **arg****, datos de entrenamiento; **función**, siendo la función de activación que se usará siendo por defecto la función *multicuadrática*; **epsilon**, una constante que aproxima la distancia promedio entre los datos con menor valor del target.

3.2.3. RNA de Regresión Generalizada (GRNN)

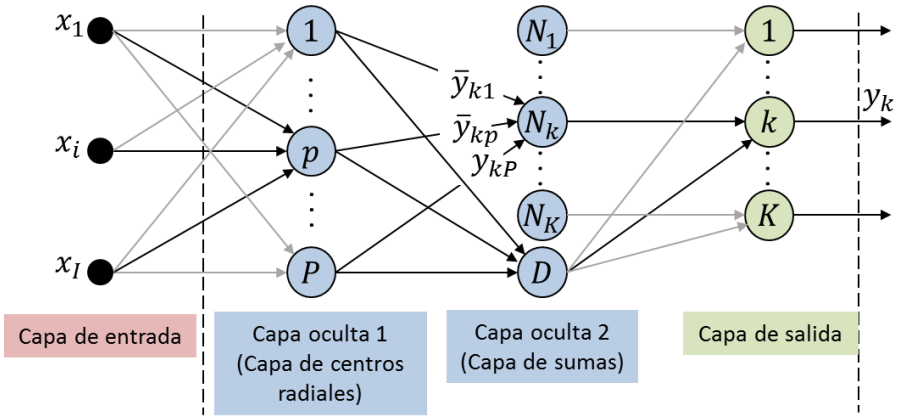


Figura 3.9: Red Neuronal Artificial de Regresión Generalizada.

La arquitectura de una red neuronal artificial de regresión generalizada (Generalized Regression Neural Network, **GRNN**) toma el modelo de la capa de centros radiales de la RBFN, como primera capa oculta, seguido a una segunda capa oculta denominada *Capa de Sumas* la cual contiene dos tipos de unidades (numerador y denominador). La capa de salida ejecuta las divisiones entre los resultado de las unidades Numerador y Denominador.

PROCESO:

- **Predicción**: Sean P los datos de entrenamiento $(\overline{\mathbf{x}}_p, \overline{\mathbf{z}}_p)$. Se establece el vector predictivo $\mathbf{y} = (y_1, \ldots, y_K)$ para un vector de entrada $\mathbf{x}$, mediante la siguiente ecuación:

$$\mathbf{y}_k = \frac{\sum_{p=1}^{P} \overline{\mathbf{y}}_{kp} \exp\left(-h(\mathbf{x}, \overline{\mathbf{x}}_p)\right)}{\sum_{p=1}^{P} \exp\left(-h(\mathbf{x}, \overline{\mathbf{x}}_p)\right)} \tag{3.19}$$

  donde w_{kp} son los pesos desde las unidades p de la primera capa culta hacia las unidades N_k de la segunda capa oculta y h se define como:

$$h(\mathbf{x}, \overline{\mathbf{x}}_p) = \sum_{i=1}^{I} \left(\frac{x_i - \overline{x}_{ip}}{\sigma} \right)^2 \tag{3.20}$$

- **Fase de entrenamiento**: El objetivo es encontrar el valor óptimo del parámetro σ mediante la minimización de la Suma Cuadrática de Errores (Error Sum of Squares, SSE), definido en (3.16).

3.2.3.1. Aplicación (Boston House-Prices)

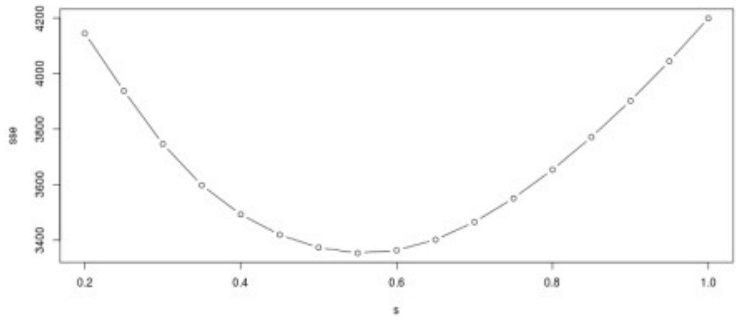
En 1978 se publicó "Hedonic prices and the demand for clean air" por Harrison y Rubinfeld, quienes proporcionaron datos que contenía información sobre las viviendas en el área de Boston, denominada **Boston Housing Dataset**. Los datos contienen 14 características, donde una de ellas es el precio siendo ésta importante a ser predicha. El código desarrollado en R nos proporciona un modelo de predicción GRNN para los precios de viviendas tomando como valores de entrada una de las características para determinarlas.

Listing 3.5: GRNN/Boston House-prices (Fuente: Statcompute, Lenguaje: R).

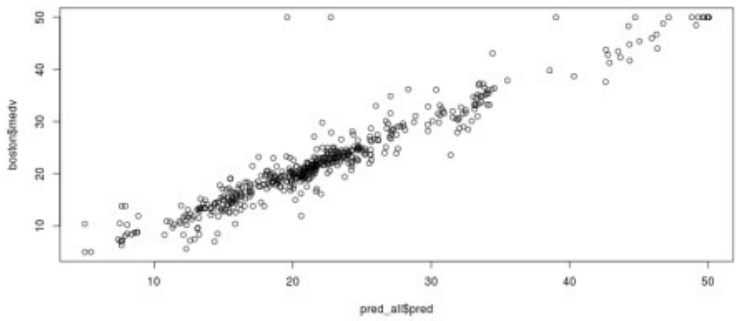
```
pkgs <- c('MASS', 'doParallel', 'foreach', 'grnn')
lapply(pkgs, require, character.only = T)
registerDoParallel(cores = 8)
data(Boston)
% Datos preprocesados
X <- Boston[-14]
st.X <- scale(X)
Y <- Boston[14]
boston <- data.frame(st.X, Y)
% Dividir la muestra de los datos
set.seed(2013)
rows <- sample(1:nrow(boston), nrow(boston) - 200)
set1 <- boston[rows, ]
set2 <- boston[-rows, ]
% Definir una funcion de calificacion GRNN
pred_grnn <- function(x, nn){
        xlst <- split(x, 1:nrow(x))
        pred <- foreach(i = xlst, .combine = rbind){
                data.frame(pred = guess(nn, as.matrix(i)), i, row.names = NULL)
        }
}
% Busqueda del valor optimo de SIGMA mediante Validacion Simple
cv <- foreach(s = seq(0.2, 1, 0.05), .combine = rbind){
        grnn <- smooth(learn(set1, variable.column = ncol(set1)), sigma = s)
        pred <- pred_grnn(set2[, -ncol(set2)], grnn)
        test.sse <- sum((set2[, ncol(set2)] - pred$pred)^2)
        data.frame(s, sse = test.sse)
}
cat("\n### SSE para validaciones ###\n")
print(cv)
jpeg('grnn_cv.jpeg', width = 800, height = 400, quality = 100)
with(cv, plot(s, sse, type = 'b'))
cat("\n### mejor valor de SIGMA con MENOR SSE ###\n")
print(best.s <- cv[cv$sse == min(cv$sse), 1])
% Calificacion de toda la data con GRNN
final_grnn <- smooth(learn(set1, variable.column = ncol(set1)), sigma = best.s)
pred_all <- pred_grnn(boston[, -ncol(set2)], final_grnn)
jpeg('grnn_fit.jpeg', width = 800, height = 400, quality = 100)
plot(pred_all$pred, boston$medv)
dev.off()
```

En listing 3.5, se extrae una única carácterística de Boston Housing Dataset asignando como targets a los precios, luego se define una función de calificación o predicción

para vectores de entrada nuevos, respecto a ese tipo de característica. Posteriormente, se realiza la búsqueda del valor óptimo σ de la ecuación (3.20) usando SSE. Habiéndose determinado lo anterior, se establece una comparación entre targets reales y valores de salida predichos por la red.



(a) $\sigma \times SSE$.



(b) *PRECIO predicho* $\times$ *PRECIO real*

Figura 3.10: Visualización gráfica para la obtención del óptimo σ y comparaciones entre los valores reales y de predicción de los precios de una casa usando GRNN.

PAQUETE grnn:
Este paquete diseñado en R, modela la red neuronal de regresión generalizada implementando el algoritmo propuesto por Specht (1991), y consta de los siguientes métodos que han sido implementandos:

- **guess(nn,X)**: Infiere el valor de una nueva observación, donde **nn** representa la red entrenada y **X** un vector de prueba.
- **learn(set, variable.column = 1)**: Crea o actualiza una GRNN, donde **set** es el marco de datos representando el conjunto de entrenamiento y **variable.column** el número de campo de la variable que por defecto es 1.
- **smooth(nn,sigma)**: Suaviza una GRNN, donde **nn** representa la red entrenada y **sigma** un escalar.

3.2.4. RNA Kohonen (SOM)

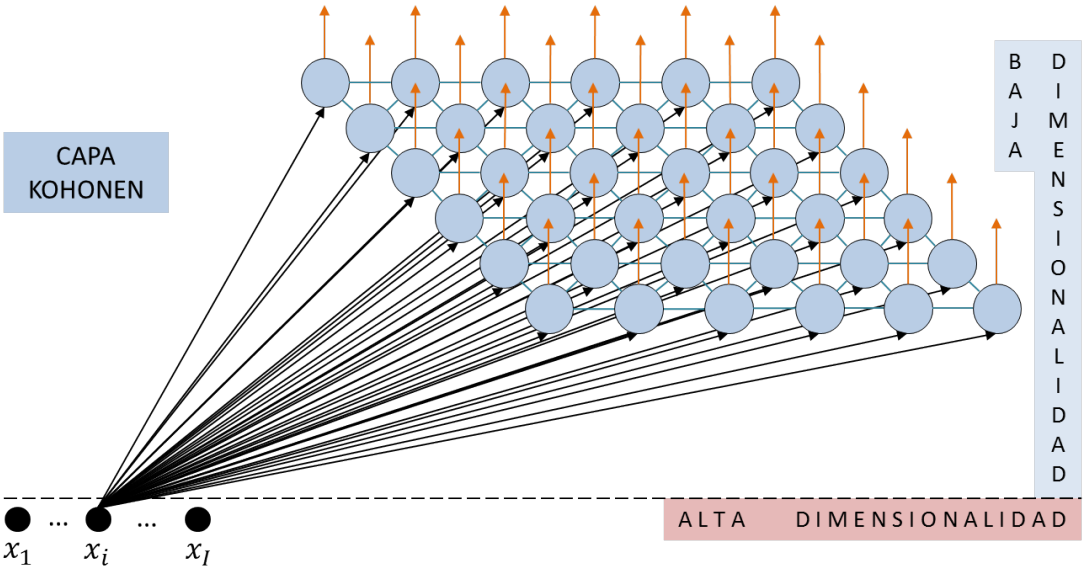


Figura 3.11: Red Neuronal Artificial Kohonen en rejilla rectangular.

La arquitectura de una red neuronal artificial Kohonen, o comúnmente denominado *Mapa auto-organizado* (Self-Organizing Map, **SOM**) es diseñada de tal forma que exista un arreglo "simétrico" entre las neuronas de una capa, siendo denominada a esta última como **Capa Kohonen**, con el objetivo clave de la reducción de un espacio de alta dimensionalidad (continua o discreta) a un espacio de menor dimensionalidad discreta. Este arreglo es denominado *rejilla* y en la mayoría es de forma rectangular (Figura 3.11) o hexagonal. SOM es una de las RNA con mayor dificultad computacional de entrenamiento.

PROCESO (en la Capa Kohonen):

1. **Competencia**: Por cada vector de entrada, una *función discriminante*[2] actúa sobre ella para computar un resultado, siendo definida como la distancia euclidiana entre tal vector y el peso sináptico que le corresponda; para una unidad j en la capa Kohonen viene dado por:

$$d_j(\mathbf{x}) = \sum_{i=1}^{D}(x_i - w_{ji})^2 \tag{3.21}$$

   donde w_{ji} es el peso sináptico entre el valor x_i hacia la unidad j.
   La unidad j que establezca el menor resultado es declarada "ganadora", cuyo análisis establece que es la unidad en la capa Kohonen cuyos pesos están más cercanos a sus valores de entrada.

[2]Función establecida en un proceso en particular para la regla de aprendizaje.

2. **Cooperación**: La unidad ganadora establece un espacio topológico para una vecindad, proporcionando una base para la cooperación de éstas, y es tomada como:

$$T_{\widehat{j}j} = \exp\left(-\frac{S_{\widehat{j}j}^2}{2\sigma^2}\right), \text{ con } \sigma(k) = \sigma(0)e^{-\frac{k}{\tau_\sigma}} \tag{3.22}$$

donde $S_{\widehat{j}j}$ es la distancia entre la unidad ganadora $\widehat{j}$ y una unidad j, y σ es el tamaño de vecindad la cual decrece con el tiempo k. Se presentan las siguientes propiedades de la topologia de vecindad: *a*) alcanza un máximo sobre la misma neurona ganadora; *b*) presenta simetría alrededor de una neurona, esto es a causa del diseño de rejilla que se estructura en la capa de Kohonen; *c*) decrece monóticamente a cero cuando $S_{\widehat{j}j} \to \infty$; *d*) Posee traslación invariante, a causa del mismo hecho por la cual existe simetría.

3. **Adaptación**: Se establece la actualización de pesos como sigue:

$$w_{ji}(k+1) = w_{ji}(k) + \underbrace{\gamma(k)(T_{\widehat{j}j}(k))(x_i - w_{ji}(k))}_{\Delta w_{ji}(k)} \quad \text{con } \gamma(k) = \gamma(0)e^{-\frac{k}{\tau_\gamma}} \tag{3.23}$$

Algoritmo 8: Mapeo Auto-organizado

Datos: $w_{ji}(0) \;\; \forall i \in \{1,\ldots,I\}, j \in \{1,\ldots,J\}$; σ_0; τ_σ; γ_0; τ_γ.

$k = 0$;
mientras $k \geq 0$ **hacer**
  Tomar $\mathbf{x}$;
  $\widehat{j} = \text{mín}\left\{d_j(\mathbf{x}) = \sum_{i=1}^{I}(x_i - w_{ji}(k))^2\right\}$;
  $\sigma(k) = \sigma(0)\exp(-k/\tau_\sigma)$;
  $\gamma(k) = \gamma(0)\exp(-k/\tau_\gamma)$;
  para $j = 1,\ldots,N$ **hacer**
    $T_{\widehat{j}j}(k) = \exp\left(-\frac{S_{\widehat{j}j}^2}{2\sigma(k)^2}\right)$;
    para $i = 1,\ldots,D$ **hacer**
      $w_{ji}(k+1) \leftarrow w_{ji}(k) + \gamma(k)T_{\widehat{j}j}(k)(x_i - w_{ji}(k))$
    fin para
  fin para
fin mientras
devolver $w_{ji}(k)$.

3.2.4.1. Aplicación (Auto-organización de datos)

Debido a que el proceso desarrollado por la red SOM, en teoría presenta una alta complejidad de entrenamiento a causa del mapeo de un espacio de dimensión alta a una de dimensión baja, se hará uso del lenguaje de programación PYTHON el cual consta de un algoritmo SOMOCLU altamente eficiente, paralelo y distribuido para entrenar tales mapas.
En listing 3.6, se generan datos con tres características (ver Figura 3.12), para luego generar con SOMOCLU, 3 planos que califiquen los datos (ver Figura 3.13). Se escogerá el plano de mejor representación y se distribuirán los datos, agregándosele adicionalmente etiquetas respectivas a cada unidad (ver Figura 3.14).

Listing 3.6: SOM (Fuente: Peter Wittek, Lenguaje: Python).

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import somoclu
%matplotlib inline
# Se generan datos aleatorios en tres categorias.
c1 = np.random.rand(50, 3)/5
c2 = (0.6, 0.1, 0.05) + np.random.rand(50, 3)/5
c3 = (0.4, 0.1, 0.7) + np.random.rand(50, 3)/5
data = np.float32(np.concatenate((c1, c2, c3)))
colors = ["red"] * 50
colors.extend(["green"] * 50)
colors.extend(["blue"] * 50)
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(data[:, 0], data[:, 1], data[:, 2], c=colors)
labels = range(150)
n_rows, n_columns = 100, 160
som = somoclu.Somoclu(n_columns, n_rows, data=data)
%time som.train()
# Planos que componen los codigos de entrenamiento.
som.view_component_planes()
# Cantidad de datos asociados a cada neurona.
som.view_umatrix(bestmatches=True, bestmatchcolors=colors, labels=labels)
```

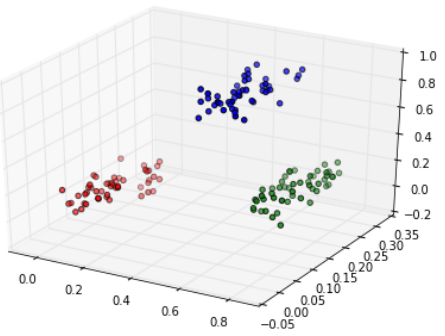
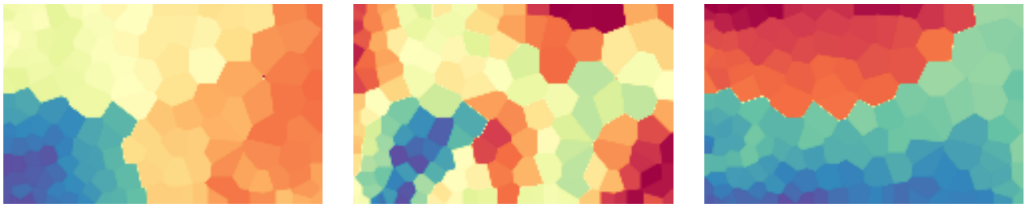


Figura 3.12: Datos de entrenamiento categorizados.



(a) Plano 1. (b) Plano 2. (c) Plano 3.

Figura 3.13: Planos de entrenamiento.

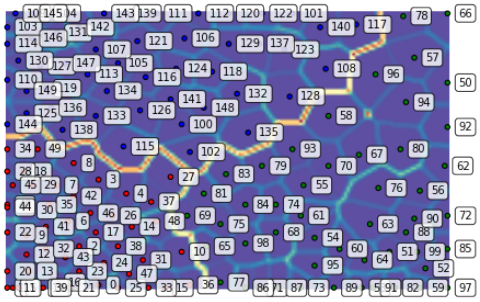


Figura 3.14: Cantidad de datos asociados a cada neurona respecto al Plano 1.

LIBRERÍA somoclu:
Somoclu es una implementación masivamente paralela de mapas auto-organizados. La topología del mapa puede resultar plana o toroidal, y la cuadrícula es rectangular o hexagonal. Consta de los siguientes métodos:

- **Somoclu(columnas, filas, data=data)**: Distribuye la **data** en una matriz con cierta cantidad de **columnas** y **filas**.
- **train()**: Módulo de entrenamiento de los datos.
- **view_component_planes()**: Se visualizan los planos de representación de los datos.
- **view_umatrix(bestmatches=True, bestmatchcolors=, labels=)**: Podemos calcular la U-Matrix, junto con las mejores unidades de coincidencia para cada punto de datos. Se colorean las unidades con las clases de los puntos de datos añadiénsoles etiquetas (labels) representadas en un rango de valores.

3.3. RNA's Retroalimentadoras

Una arquitectura de red neuronal artificial retroalimentadora, o también llamada recurrente (Recurrent Artificial Neural Network, **RNN**) presenta una topología de tipo cíclica, es decir que existen neuronas en la red que al activarse por primera vez, y ejecutar un resultado propia de ésta, pueden volver a activarse a causa de un mecanismo recurrente, diferenciándose de esta forma de una red alimentadora. Si el mecanismo es recurrente en cada par de neuronas se dice que es una red neuronal artificial totalmente recurrente[3]. La condición que se plantea es la siguiente:

"*Las informaciones fluyen y se complementan unas con otras no solo en una dirección, sino con algunos retrocesos.*"

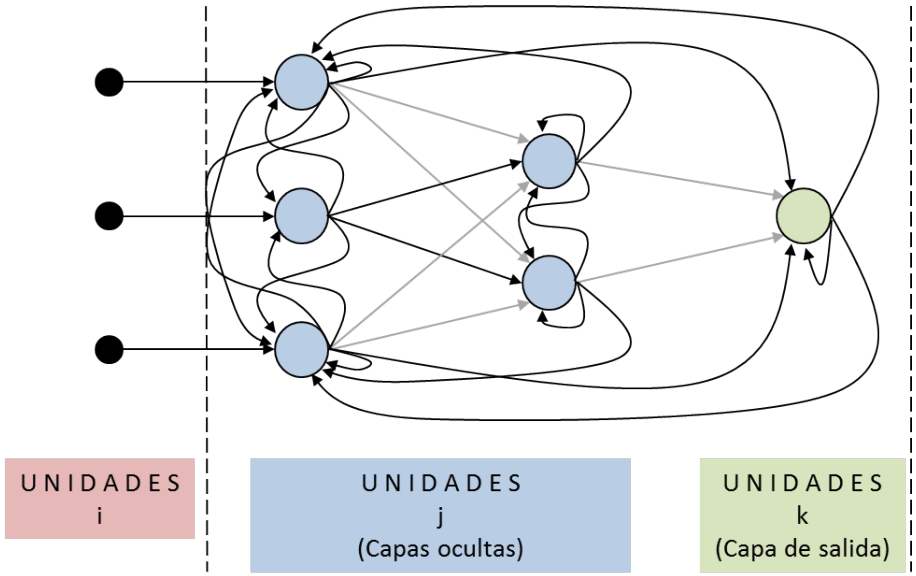


Figura 3.15: Arquitectura de una RNA Totalmente Recurrente.

[3]RNA cuyas unidades se encuentran totalmente conectadas entre sí, tanto de ida como de vuelta.

3.3.1. RNA Hopfield

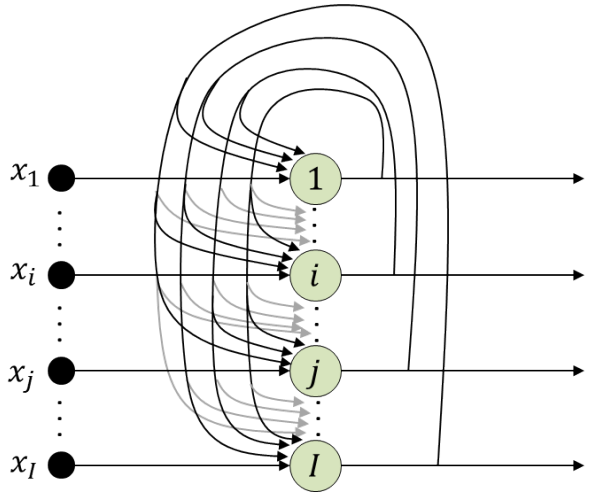


Figura 3.16: Red Neuronal Artificial Hopfield.

La arquitectura de una RNA Hopfield fue inspirada por las propiedades de las memorias asociadas en el cerebro humano. Esta red es diseñada para almacenar uno o más vectores de salida estables, siendo éstos los que representan señales en la memoria interna de la RNA para cuando vectores similares sean proporcionados en la misma. La red Hopfield posee la propiedad de recurrencia simétrica, es decir $w_{ji} = w_{ij}$, además de que no existe recursividad de una unidad sobre sí misma, es decir $w_{ii} = 0$.

PROCESO:

1. **Almacenamiento**: Los vectores de entrada son componentes del espacio H de Hamming[4], esto es $\mathbf{x} \in H^I$. Teniéndose P datos de entrenamiento $(\overline{\mathbf{x}}_p, \overline{\mathbf{z}}_p)$, P estará limitado bajo la siguiente condición: $P < 0.15I$.

2. **Pesos Sinápticos**: La actualización de los pesos sigue la regla de Hebb (1.13) y se organizan en la siguiente matriz de correlación:

$$W = (w_{ij}) \quad \text{donde} \begin{cases} w_{ij} = w_{ji} = \sum_{p=1}^{P} x_{ip} x_{jp}. \\ w_{ii} = 0. \end{cases} \tag{3.24}$$

3. **Propagación**: La activación es calculada y transferida a la salida mediante la función de paso:

$$y_i(k) = \sum_{j=1}^{I} w_{ij} f(y_j(k-1)), \quad \text{donde } f(y_i(k)) = \begin{cases} 1 & , \ y_i(k) \geq 0 \\ -1 & , \ y_i(k) < 0 \end{cases} \tag{3.25}$$

[4]Espacio que posee solo los elementos -1 y 1.

4. **Fase de recuperación**: El segundo objetivo es recordar un vector de entrenamiento, si se conoce un vector de prueba x_{test}, el cual es generalmente un vector de entrenamiento con ruido.

5. **Búsqueda**: Teniéndose W, se buscará el correspondiente vector de entrenamiento x_{test} usando el Algoritmo 9.

Algoritmo 9: Hopfield

Datos: $I(x)$, $x_{test}(1)$;
$W = \sum_{k=1}^{N} x_k x_k^T$, $w_{ii} = 0$.
$t = 1$;
mientras *No más cambios en* $x_{test}(t+1)$ **hacer**
  $net_j(t) = \langle w_j, x_{test}(t) \rangle$, $j = \overline{1, D}$;/* w_j es la columna j de W. */
  si $net_j(t) > 0$ **entonces**
    $x_{test}(t+1) = +1$;
  fin si
  si $net_j(t) < 0$ **entonces**
    $x_{test}(t+1) = -1$;
  en otro caso
    $x_{test}(t+1) = x_{test}(t)$;
  fin si
fin mientras
devolver x_{test}

3.3.1.1. Aplicación (Reconocimiento de patrones)

Se implementará un algoritmo desarrollado en PYTHON con uso de la librería hopfieldnet.net para el diseño de la red Hopfield que sea capaz de reconocer las letras A, B y C, usando un aprendizaje Hebbiano proveniente de la librería **hopfieldnet.trainers**. En listing 3.7, se crea inicialmente los patrones de entrenamiento para las letras A, B y C, los cuales servirán como puntos de comparación y reconocimiento para el patrón que genere un dato de prueba en el modelo de red Hopfield, siendo esta red creada usando la función **HopfieldNetwork()** y entrenada usando la función **hebbian_training()**. Teniéndose la red entrenada se prosigue en la comparación de datos de prueba usando **run()**, donde se obteniene como resultado los patrones de entrenamiento en donde exista un reconocimiento, en este caso una de las tres letras que mejor reconozca a cada una de las letras que se ponen a prueba. Posteriormente se visualiza el reconocimiento de patrones en Figura 3.17.

Listing 3.7: Hopfield/Reconocimiento de caracteres (Fuente: Panagiotis Matigakis, Lenguaje: Python).

```
1 from random import randint
2 import numpy as np
3 from matplotlib import pyplot as plt
```

```
from hopfieldnet.net import  HopfieldNetwork
from hopfieldnet.trainers import hebbian_training
#Crear los patrones de entrenamiento
a_pattern = np.array([[0, 0, 1, 0, 0],
                      [0, 1, 0, 1, 0],
                      [1, 0, 0, 0, 1],
                      [1, 1, 1, 1, 1],
                      [1, 0, 0, 0, 1],
                      [1, 0, 0, 0, 1],
                      [1, 0, 0, 0, 1]])
b_pattern = np.array([[1, 1, 1, 1, 0],
                      [1, 0, 0, 0, 1],
                      [1, 0, 0, 0, 1],
                      [1, 1, 1, 1, 0],
                      [1, 0, 0, 0, 1],
                      [1, 0, 0, 0, 1],
                      [1, 1, 1, 1, 0]])
c_pattern = np.array([[1, 1, 1, 1, 1],
                      [1, 0, 0, 0, 0],
                      [1, 0, 0, 0, 0],
                      [1, 0, 0, 0, 0],
                      [1, 0, 0, 0, 0],
                      [1, 0, 0, 0, 0],
                      [1, 1, 1, 1, 1]])
a_pattern *= 2
a_pattern -= 1
b_pattern *= 2
b_pattern -= 1
c_pattern *= 2
c_pattern -= 1
input_patterns = np.array([a_pattern.flatten(), b_pattern.flatten(), c_pattern.flatten()])
# Crear la RNA y entrenarla usando los patrones de entrenamiento.
network = HopfieldNetwork(35)
hebbian_training(network, input_patterns)
# Crear los patrones de prueba usando los patrones de entrenamiento.
a_test =  a_pattern.flatten()
for i in range(4):
        p = randint(0, 34)
        a_test[p] *= -1
a_result = network.run(a_test)
a_result.shape = (7, 5)
a_test.shape = (7, 5)
b_test =  b_pattern.flatten()
for i in range(4):
        p = randint(0, 34)
        b_test[p] *= -1
b_result = network.run(b_test)
b_result.shape = (7, 5)
b_test.shape = (7, 5)
c_test =  c_pattern.flatten()
for i in range(4):
        p = randint(0, 34)
        c_test[p] *= -1
c_result = network.run(c_test)
c_result.shape = (7, 5)
c_test.shape = (7, 5)
# Mostrar los resultados
```

```
plt.subplot(3, 2, 1)
plt.imshow(a_test, interpolation="nearest")
plt.subplot(3, 2, 2)
plt.imshow(a_result, interpolation="nearest")
plt.subplot(3, 2, 3)
plt.imshow(b_test, interpolation="nearest")
plt.subplot(3, 2, 4)
plt.imshow(b_result, interpolation="nearest")
plt.subplot(3, 2, 5)
plt.imshow(c_test, interpolation="nearest")
plt.subplot(3, 2, 6)
plt.imshow(c_result, interpolation="nearest")
plt.show()
```

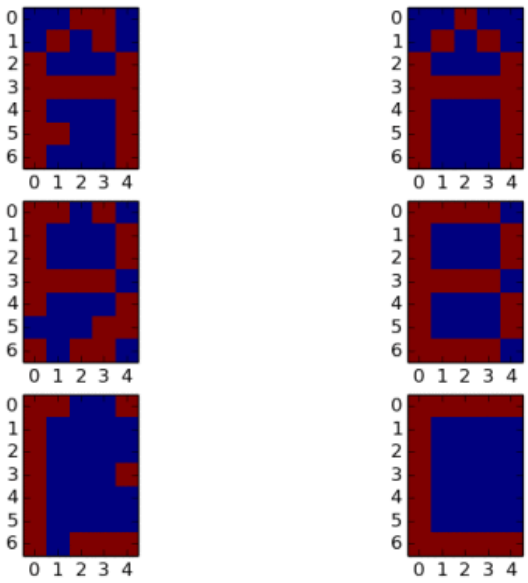


Figura 3.17: La primera columna muestra datos con ruido; y la segunda columna, la respuesta efectuada por red neuronal. Las filas son representaciones de las letras A,B y C; como se observa existe un reconocimiento de caracteres.

FUNCIÓN HopfieldNetwork(n):
Esta función pertenece a la librería **hopfieldnet.net**, y tiene como parámetro a la cantidad **n**, siendo ésta la cantidad de unidades de un diseño de red Hopfield.

- **run(x_test)**: Este modulo compara un valor de prueba **x_test** con los patrones de entrenamiento de la red Hopfield, ejecutando el patrón de reconocimiento.

FUNCIÓN hebbian_training(red, patrones_de_entrenamiento):
Esta función correspondiente a la librería **hopfieldnet.trainers**, toma como parámetros a la **red** Hopfield creada usando la función HopfieldNetwork(n) y a los **patrones de entrenamiento** creadas como cadenas.

3.3.2. RNA de Elman y Jordan

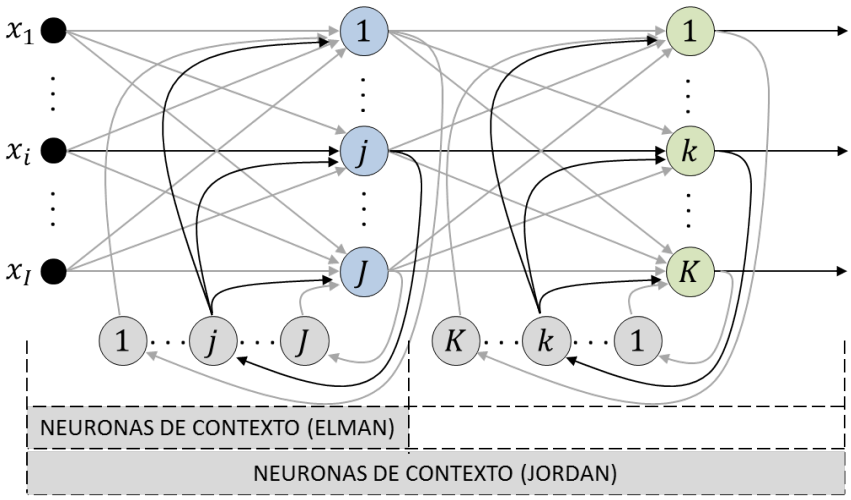


Figura 3.18: Red Neuronal Artificial Elman (Proceso recurrente en capa oculta). Red Neuronal Artificial Jordan (Proceso recurrente en capa oculta y de salida).

La arquitectura de una RNA de Elman presenta tres capas, en donde su capa oculta mantiene un proceso recurrente con transición en otras neuronas denominadas *neuronas de contexto*. Comúnmente las funciones de activación son sigmoidales en la capa oculta y lineales en la capa de salida, aproximando en conjunto cualquier función con relativa exactitud dependiendo la suficiencia de neuronas en la capa oculta. La arquitectura de una RNA de Jordan es similar a la de Elman, a diferencia que adicionan neuronas de contexto en su capa de salida. Estos dos tipos de RNA tienen memoria para detectar y generar patrones variables en el tiempo.

PROCESO (RNA DE ELMAN):
Se entrenará una red con un solo valor de salida.

1. **Flujo de activación**: Sean T los datos de entrenamiento, $(\overline{\mathbf{x}}(t), \overline{z}(t))$; u_j, v_j las entradas y salidas correspondientes a la unidad oculta j; y v_j^c las salidas para una unidad de contexto j; se siguen las siguientes ecuaciones:

$$\begin{aligned} u_j(t) &= \mathbf{w}_j^x(t-1)\overline{\mathbf{x}}(t) + \sum_{l=1}^{J} w_{jl}(t-1)v_l^c(t) \\ v_j(t) = f\left(u_j(t)\right), \;\; v_j^c(t) &= v_j(t-1), \;\; \overline{y}(t) = \sum_{j=1}^{J} w_j^y(t-1)v_j(t) \end{aligned} \tag{3.26}$$

   donde $\mathbf{w}_j^x$ es el vector de pesos desde el vector de entrada a la red y una unidad oculta j; w_{jl} es el peso desde la unidad de contexto l hacia la unidad oculta j; w_j^y el peso desde la unidad oculta j hacia la unidad de salida; $\overline{y}$ el valor de salida de la red; y f la función de activación sigmoidal en la capa oculta.

2. **Espacio de estados**: La descripción del espacio de estados de un sistema dinámico lineal de orden J, se da por el retraso del vector de entrada hacia la capa de entrada, y reemplaza (3.26) por las siguientes ecuaciones:

$$\begin{aligned} u_j(t) &= \mathbf{w}_j^x(t-1)\overline{\mathbf{x}}(t-1) + \sum_{l=1}^{J} w_{jl}(t-1)v_l(t-1) \\ v_j(t) &= u_j(t); \;\; \overline{y}(t) = \sum_{j=1}^{J} w_j^y(t-1)v_j(t) \end{aligned} \tag{3.27}$$

El vector $\mathbf{v}^c(t) = \{v_i^c(t)\}$ es $\mathbf{v}(t-1)$, la cual a su vez viene a ser una función de $\mathbf{w}(t-2)v(t-2) + \mathbf{w}^x(t-2)x(t-2)$.

3. **Errores**: Siendo el error medio cuadrático:

$$E(t) \;\; = \;\; \frac{1}{2}(\overline{z}(t) - \overline{y}(t))^2, \tag{3.28}$$

los pesos son modificados en cada tiempo t, determinado los siguiente gradientes de error:

$$\frac{\partial E(t)}{\partial w_j^y(t-1)} \;\; = \;\; -(\overline{z}(t) - \overline{y}(t))v_j(t) \tag{3.29}$$

$$\frac{\partial E(t)}{\partial \mathbf{w}_j^x(t-1)} \;\; = \;\; -(\overline{z}(t) - \overline{y}(t))w_j^y(t-1)f_{u_j}\overline{\mathbf{x}}(t) \tag{3.30}$$

$$\frac{\partial E(t)}{\partial w_{jl}(t-1)} \;\; = \;\; -(\overline{z}(t) - \overline{y}(t))w_j^y(t-1)\frac{\partial v_j(t)}{\partial w_{jl}(t-1)} \tag{3.31}$$

donde, asumiendo un cambio mínimo de pesos en cada iteración $\partial w_{jl}(t-1) \equiv \partial w_{jl}(t-2)$, tenemos que:

$$\frac{\partial v_j(t)}{\partial w_{jl}(t-1)} = f_{u_j}\left\{ v_j(t-1) + \sum_{m=1}^{J} w_{jm}\frac{\partial v_m(t-1)}{\partial w_{jl}(t-2)} \right\} \tag{3.32}$$

4. **Actualicación de pesos**: La actualización de los pesos sinápticos, mediante la ténica del descenso de gradiente, viene dada por:

$$\Delta w = -\eta\frac{\partial E_k}{\partial w} \tag{3.33}$$

obteniendo las siguientes ecuaciones:

$$\Delta w_j^y \;\; = \;\; \eta\left(\overline{z}(t) - \overline{y}(t)\right)v_j(t) \tag{3.34}$$

$$\Delta \mathbf{w}_j^x \;\; = \;\; \eta\left(\overline{z}(t) - \overline{y}(t)\right)w_i^y(t-1)f_{u_j}\overline{\mathbf{x}}(t) \tag{3.35}$$

$$\Delta w_{jl} \;\; = \;\; \eta\left(\overline{z}(t) - \overline{y}(t)\right)w_i^y(t-1)\frac{\partial v_j(t)}{\partial w_{jl}(t-1)} \tag{3.36}$$

donde

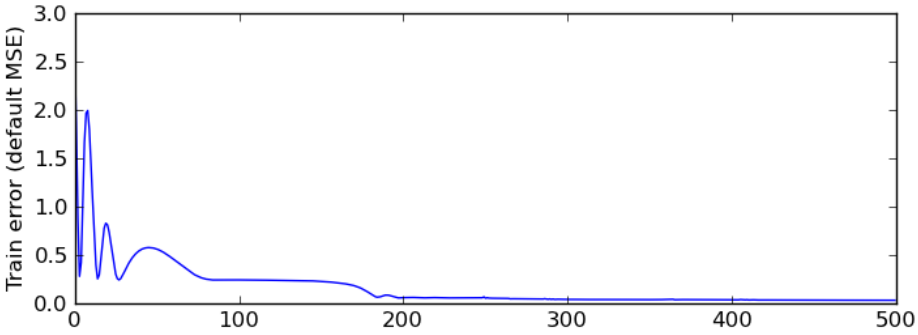
$$f_{u_j} = \frac{\partial f}{\partial u_j} = \left(1 - f^2(u_j)\right) \tag{3.37}$$

3.3.2.1. Aplicación (Detección de amplitudes)

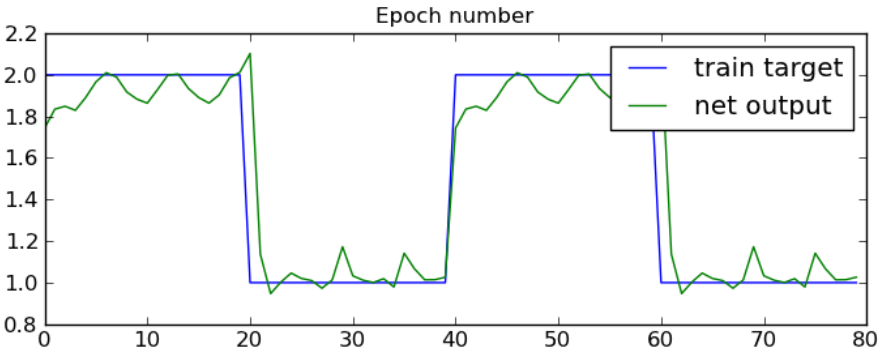
En listing 3.8, se crean datos de entrenamiento con valores de entrada no negativos, posean targets de 1 y 2 en intervalos de 20 en 20, comenzando por el mayor valor. Se crea una red de Elman, usando la función **net.newelm()** con dos capas, la primera con 10 unidades y con funciones de activación tangencial sigmoidal, y la segunda con 1 unidad con función de activación lineal. Se establece el rango [-0.1,0.1] de valores para los pesos sinápticos y los umbrales en cada capa. Se entrena la red mediante **train()** y luego se ejecutarán los valores de salida sobre los datos de entrenamiento usando **sim()**. Se visualiza en Figura 3.19.a) como disminuye el error de entrenamiento conforme se incrementa el número de iteraciones en el entrenamiento además en Figura 3.19.b) se observa la comparación entre los targets y valores de salida de la red sobre los datos de entrenamiento.

Listing 3.8: ELMAN/Detectar amplitudes (Fuente: pythonhosted.org, Lenguaje: Python).

```
import neurolab as nl
import numpy as np
# Crear datos de entrenamiento
i1 = np.sin(np.arange(0, 20))
i2 = np.sin(np.arange(0, 20)) * 2
# Targets
t1 = np.ones([1, 20])
t2 = np.ones([1, 20]) * 2
# Organizando en forma matricial
input = np.array([i1, i2, i1, i2]).reshape(20 * 4, 1)
target = np.array([t1, t2, t1, t2]).reshape(20 * 4, 1)
# Crear la red con 2 capas ocultas
net = nl.net.newelm([[-2, 2]], [10, 1], [nl.trans.TanSig(), nl.trans.PureLin()])
# Establecer funciones
net.layers[0].initf = nl.init.InitRand([-0.1, 0.1], 'wb')
net.layers[1].initf= nl.init.InitRand([-0.1, 0.1], 'wb')
# Entrenamiento de la red
error = net.train(input, target, epochs=500, show=100, goal=0.01)
# Simulacion de la red
output = net.sim(input)
# Ploteo de resultados
import pylab as pl
pl.subplot(211)
pl.plot(error)
pl.xlabel('Epoch number')
pl.ylabel('Train error (default MSE)')
pl.subplot(212)
pl.plot(target.reshape(80))
pl.plot(output.reshape(80))
pl.legend(['train target', 'net output'])
pl.show()
```



(a) Iteraciones vs Error.



(b) Comparación entre targets y salida de la RNA.

Figura 3.19: Resultados obtenidos en la aplicación de una red de Elman para detectar amplitudes.

FUNCIÓN neurolab.net.newelm(minmax, tamaño, activ):
Esta función diseña una red Elman y corresponde a la librería **neurolab**, presentando los siguientes parámetros: **minmax**, una lista que representa el rango de un valor de entrada; **size**, una lista cuyo tamaño representa la cantidad de capas a excepción de la capa de entrada y cada elemento de la lista es el número de neuronas correspondiente a cada capa; **activ**, una lista cuyos elementos corresponden a los tipos de funciones de activación respecto a cada capa generada por el parámetro *size*, los más comunes son *trans.TanSig()* y *trans.PureLin()* siendo las funciones tangencial sigmoidal y lineal respectivamente. Se establecen los siguientes métodos:

- **layer[i].initf = init.InitRand(lista, 'wb')**: Con esto daremos a **lista** un rango de valores que pueden tomar los pesos **w** y el umbral **b** de las unidades de la capa oculta **i**.
- **train(input, target, epochs=n, show=m, goal=epsilon)**: Entrena la red usando los parámetros: **input** y **targets** de los datos de entrenamiento; **epoch**, iteraciones llevadas a cabo; **show**, muestra el intervalo de iteraciones cuando se ejecute un ploteo de este método; **goal**, es la cota de error en el entrenamiento.
- **sim(input)**: Genera los valores de salida obtenidos por la red sobre los vectores de entrada de los datos de entrenamiento.

3.3.3. RNA con Memoria de Corto y Largo Plazo (LSTM)

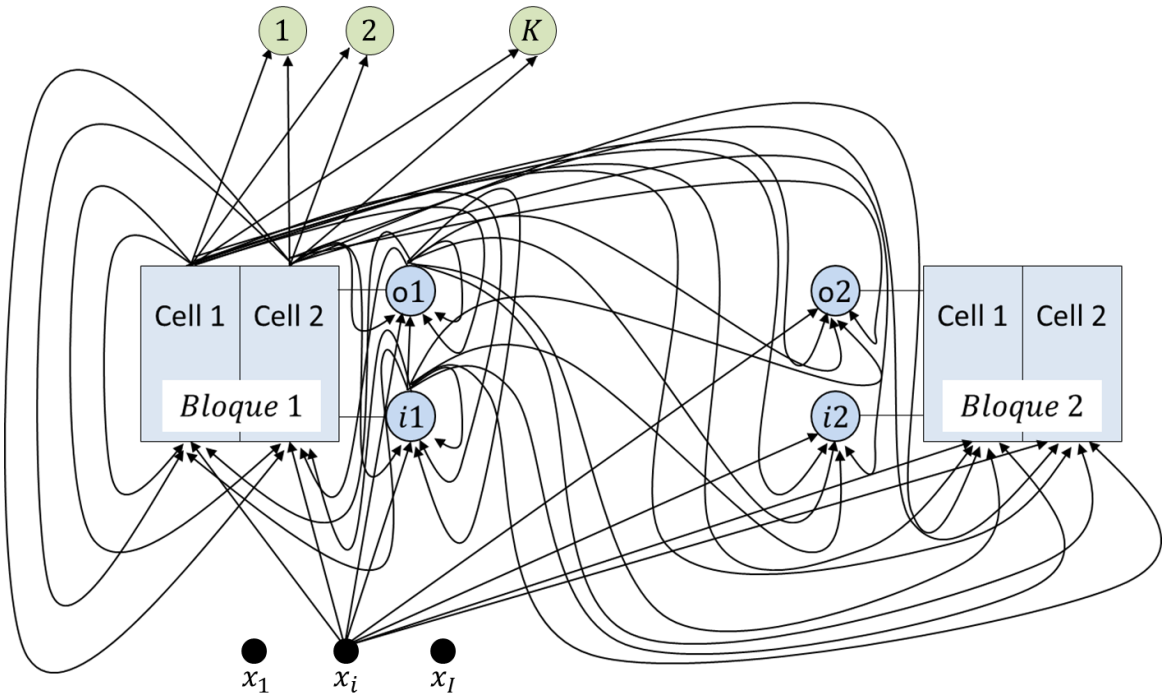


Figura 3.20: Red Neuronal Artificial con Memoria de Corto y Largo Plazo.

La arquitectura de una RNA con memoria de corto y largo plazo (Long Short Term Memory, **LSTM**) es construida con bloques de memoria, los cuales a su vez están compuestos por celdas de memoria, que otorgan a la red la capacidad de recordar valores por cualquier longitud de tiempo, y ciertas unidades que funcionan como puertas tanto de entrada, denominadas *input gates*, como de salida, denominadas *output gates*, que determinan cúando estas entradas o salidas son lo suficientemente importantes como para recordarlas. En conjunto, LSTM puede aprender de su experiencia para procesar, clasificar y predecir series de tiempo.

Celda de memoria (estándar):
Una celda de memoria (Figura 3.19) posee en su interior una unidad autorrecurrente, llamada *Constant Error Carousel (CEC)*, la cual ejecuta señales mediante una función lineal donde su activación se denomina *estado de la celda*. La recurrencia del CEC tiene como objetivo ayudar en la toma de decisiones y es protegida por el flujo de activación y el flujo hacia atrás de los errores. Un bloque de memoria, o bloque LSTM, es un conjunto de celdas de memoria conectado con un input gate, la cual la proteje de algún tipo de perturbación generado por valores de entrada irrelevantes, y un output gate, que proteje otras unidades de perturbaciones generadas por contenidos de memoria irrelevantes.

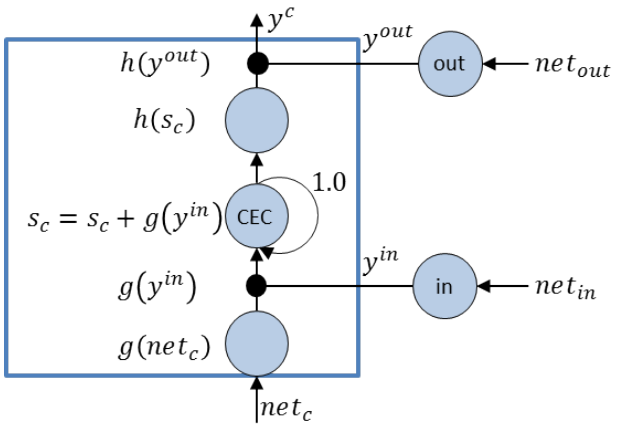


Figura 3.21: Bloque LSTM con una sola celda de memoria.

Notaciones 1:
net_c: vector de entrada a la celda c; net_{in},net_{out}: vectores entradas al input gate y output gate respectivamente; k: representa una unidad de salida; j: representa al j-ésimo bloque LSTM; c_j^v: representa la unidad c de la v-ésima celda del j-ésimo bloque LSTM; w_{lm}: peso sináptico desde la unidad m hacia la unidad l; s_c: estado de la celda; t: tiempo discreto; f, g, h: funciones sigmoidales logísticas de rango $[0, 1]$ para los gates, $[-2, 2]$ y $[-1, 1]$ respectivamente dados como:

$$f(x) = \frac{1}{1+e^{-x}}; \quad g(x) = \frac{4}{1+e^{-x}} - 2; \quad h(x) = \frac{2}{1+e^{-x}} - 1. \tag{3.38}$$

Proceso Estándar:

1. **Activación**: La activación en el input gate, output gate y una unidad k, es calculada mediante las siguientes ecuaciones:

$$net_l(t) = \sum_m w_{lm} y^m(t-1); \quad y^l(t) = f_l(net_l(t)) \text{ donde } l = in_j, out_j. \tag{3.39}$$

   La activación de la unidad recurrente c_j^v, a causa de su función de activación lineal, viene dada como:

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m} y^m(t-1) \tag{3.40}$$

   la cual es luego mapeada por una función g.

2. **Estado de la celda**: El estado interno de la celda es calculado por:

$$\begin{aligned} s_{c_j^v}(0) &= 0, \\ s_{c_j^v}(t) &= s_{c_j^v}(t-1) + y^{in_j}(t) g\left(net_{c_j^v}(t)\right) \quad \forall t > 0. \end{aligned} \tag{3.41}$$

3. **Salida de la celda**: El valor de salida correspondiente a una celda viene representada como:

$$y^{c_j^v} = y^{out_j}(t) h\left(s_{c_j^v}(t)\right). \tag{3.42}$$

Celda de memoria con Forget Gates:
La implementación de una nueva unidad denominada *Forget gate* en cada celda fue a causa de la saturación provocada por la función h correspondiente a la arquitectura de un LSTM estándar. Esta saturación hacía que, por mecanismos secuenciales, la salida en una celda sea igual a la activación del output gate pues evitando de esa forma que la celda funcione como una memoria. El forget gate, funciona como una unidad reiniciadora sobre celdas que contienen informaciones inutilizables ejerciendo un proceso de memorización y olvido en cada nueva secuencia iniciado en la red.

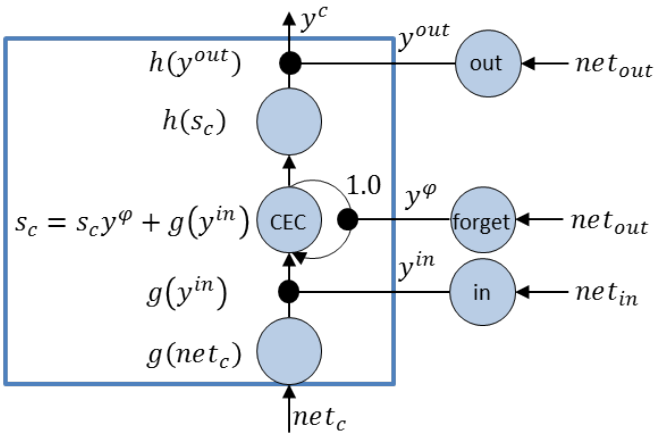


Figura 3.22: Bloque LSTM con una sola celda de memoria.

Notaciones 2:
net_{φ_j}: entrada hacia el forget gate respecto al j-ésimo bloque LSTM; $\overset{tr}{=}$: truncamiento de error; t^k:targets; e_k: error externo respecto a la unidad de salida k, η: tasa de aprendizaje.
Proceso Extendido:

1. **Forward pass**: La ativación y^{φ} del forget gate es calculado de forma análoga que (3.45) y viene dado como:

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m} y^m(t-1); \;\; y^{\varphi_j}(t) = f_{\varphi_j}\left(net_{\varphi_j}(t)\right) \tag{3.43}$$

El estado de la celda vendría luego dado por:

$$\begin{aligned} s_{c_j^v}(0) &= 0, \\ s_{c_j^v}(t) &= y^{\varphi_j}(t) s_{c_j^v}(t-1) + y^{in_j}(t) g\left(net_{c_j^v}(t)\right) \;\; \forall t > 0. \end{aligned} \tag{3.44}$$

2. **Backward pass**: La función objetivo es:

$$E(t) = \frac{1}{2}\sum_k e_k^2(t); \quad e_k(t) := t^k(t) - y^k(t). \tag{3.45}$$

Esta función es minimizada por descenso de gradiente con las actualizaciones siguientes:

$$\Delta w_{lm} = \eta \underbrace{\frac{\partial y^l(t)}{\partial net_l(t)}\left[\sum_k \frac{\partial y^k(t)}{\partial y^l(t)} e_k(t)\right]}_{\delta_l(t)} y^m(t-1). \tag{3.46}$$

donde para $l = k$, y su contribución a la v-ésima unidad, se tiene:

$$\delta_k(t) = f_k'\left(net_k(t)\right) e_k(t); \tag{3.47}$$

$$\delta^v_{out_j}(t) = f'_{out_j}\left(net_{out_j}(t)\right) h\left(s_{c_j^v}(t)\right)\left[\sum_k w_{kc_j^v}\delta_k(t)\right]. \tag{3.48}$$

Las actualización de pesos es reformulada de la siguiente forma:

$$\Delta w_{c_j^v m}(t) = \eta e_{s_{c_j^v}}(t)\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \tag{3.49}$$

$$\Delta w_{lm}(t) = \eta \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t)\frac{\partial s_{c_j^v}(t)}{\partial w_{lm}}; \quad l = \varphi, in. \tag{3.50}$$

donde

$$\frac{\partial s_{c_j^v}(0)}{\partial w_{c_j^v m}} = 0, \quad \frac{\partial s_{c_j^v}(0)}{\partial w_{in_j m}} = 0, \quad \frac{\partial s_{c_j^v}(0)}{\partial w_{out_j m}} = 0,$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}} y^{\varphi_j}(t) + g'\left(net_{c_j^v}\right) y^{in_j}(t) y^m(t-1), \tag{3.51}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}} y^{\varphi_j}(t) + g\left(net_{c_j^v}\right) f'_{in_j}\left(net_{in_j}(t)\right) y^m(t-1), \tag{3.52}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{out_j m}} = \frac{\partial s_{c_j^v}(t)}{\partial w_{out_j m}} y^{\varphi_j}(t) + h\left(s_{c_j^v}\right) f'_{\varphi_j}\left(net_{\varphi_j}(t)\right) y^m(t-1) \tag{3.53}$$

y además

$$e_{s_{c_j^v}}(t) = y^{out_j}(t) h'\left(s_{c_j^v}(t)\right). \tag{3.54}$$

Algoritmo 10: LSTM extendido

mientras *Criterio de error* **hacer**

`/* FORWARD PASS:` `*/`

`/* capa de entrada` $\rightarrow$ `Capa oculta` `*/`

Reiniciar todos los valores net;

$net_l(t) = \sum_m w_{lm} y^m(t-1),\ l \in \{in_j,\ \varphi_j, out_j, c_j^v\}$;

`/* Activación en la capa oculta` `*/`

$y^l(t) = f_l(net_l(t)),\ l \in \{in_j,\ \varphi_j, out_j\}$;

$s_{c_j^v}(0) = 0;\ \ s_{c_j^v}(t) = y^{\varphi_j}(t) s_{c_j^v}(t-1) + y^{in_j}(t) g(net_{c_j^v}(t))$;

$y^{c_j^v}(t) = y^{out_j}(t) h\left(s_{c_j^v}(t)\right)$;

`/*` net_{in} `y activación en las unidades de salida` `*/`

$net_k(t) = \sum_m w_{km} y^m(t-1),\ y^k(t) = f_k\left(net_k(t)\right)$;

`/* Derivadas` `*/`

$dS_{lm}^{jv}(t) = \dfrac{\partial s_{c_j^v}}{\partial w_{lm}},\ \bar{l} \in \{\varphi,\ in,\ c\}$;

$dS_{in,m}^{jv}(0) = 0;\ dS_{\varphi m}^{jv}(0) = 0;\ dS_{cm}^{jv}(0) = 0$

$dS_{in,m}^{jv}(t) = dS_{in,m}^{jv}(t-1) y^{\varphi_j}(t) + g\left(net_{c_j^v}(t)\right) f'_{in_j}\left(net_{in_j}(t)\right) y^m(t-1)$;

$dS_{\varphi m}^{jv}(t) = dS_{\varphi m}^{jv}(t-1) y^{\varphi_j}(t) + g\left(s_{c_j^v}(t)\right) f'_{\varphi_j}\left(net_{\varphi_j}(t)\right) y^m(t-1)$;

$dS_{cm}^{jv}(t) = dS_{cm}^{jv}(t-1) y^{\varphi_j}(t) + g'\left(net_{c_j^v}\right) y^{in_j}(t) y^m(t-1)$;

`/* BACKWARD PASS:` `*/`

`/* errores y` δ`'s` `*/`

$e_k(t) := t^k(t) - y^k(t)$;

$\delta_k(t) = f'_k\left(net_k(t)\right) e_k(t)$;

$$\delta_{out_j}(t) = f'_{out_j}\left(net_{out_j}(t)\right)\left(\sum_{v=1}^{S_j} h\left(s_{c_j^v}(t)\right) \sum_k w_{kc_j^v} \delta_k(t)\right);$$

$$e_{s_{c_j^v}}(t) = y^{out_j}(t) h'\left(s_{c_j^v}(t)\right)\left(\sum_k w_{kc_j^v} \delta_k(t)\right).$$

`/* Actualización de pesos` `*/`

$\Delta w_{lm}(t) = \eta \delta_l(t) y^m(t-1),\ l \in \{k,\ i,\ out\}$;

$\Delta w_{lm}(t) = \eta \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t) dS_{lm}^{jv}(t),\ \bar{l} \in \{in,\ \varphi\}$;

$\Delta w_{lm}(t) = \eta e_{s_{c_j^v}}(t) dS_{lm}^{jv}(t)$;

3.3.3.1. Aplicación (Series de tiempo)

La siguiente aplicación consta en predecir el número de pasajeros de una línea aérea internacional contando con los datos de los meses pasados hasta la actualidad, por tanto es un problema con series de tiempo por ser los datos una secuencia de observaciones medidas cronológicamente.
En listing 3.9, se desarrolla un código en el lenguaje de programación PYTHON para el modelamiento de una red LSTM y aplicado al problema planteado, en donde las librerías de uso especial son keras, pandas, sklearn.preprocessing y sklearn.metrics. En primera instancia, se carga el marco de datos de un archivo llamado *'international-airline-passengers.csv'*, el cual consta con la base de datos esencial, posteriormente se extrae la matriz de datos convirtiéndolos luego de valores enteros a valores de coma flotante, más adecuados al modelamiento que se llevará a cabo. Luego se normalizarán estos datos usando **MinMaxScaler**, a una escala de 0 a 1, debido a la sensibilidad que puede presentar la red LSTM a ciertas escalas. Habiéndose normalizado los datos, se procederá a dividirlos en datos de entrenamiento y datos de prueba, en un 67 % y 33 % respectivamente. Se define una función **create_dataset** para crear dos subconjuntos de datos, sean **X** e **Y**, en donde el primero es el número de pasajeros en un tiempo t y el segundo en un tiempo $t + 1$; esta función será evaluada primero sobre los datos de entrenamiento diviéndola en los subconjuntos **trainX** y **trainY**, y de igual forma sobre los datos de prueba obteniéndose **testX** y **testY**. La red LSTM espera que los datos de entrada mantengan una estructura [muestras, pasos del tiempo, características] por lo que se reordenan los subconjuntos de entrenamiento y de prueba usando **numpy.reshape**. Ahora se diseña y ajusta la red LSTM al problema usando los datos de entrenamiento; esta red presenta una capa de entrada con 1 solo valor de entrada, una capa oculta con 4 bloques LSTM presentando unidades con funciones de activación sigmoidal, y una única unidad de salida. Se establecerán las predicciones para los datos de entrenamiento y los datos de prueba; posteriormente, para calcularse el error, en este caso el RECM (Raíz del Error Cuadrático Medio) es necesario antes el uso de **inverse_transform** con el fin de mantener las mismas unidades que los datos originales antes de su normalización, es decir miles de pasajeros al mes. Por último, se visualiza en Figura 3.23 el ploteo de los datos originales y los predichos tanto para los datos de entrenamiento como para los datos de prueba.

Listing 3.9: LSTM/Pasajeros línea aérea internacional (Fuente: Jason Brownlee, Lenguaje: Python).

```
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

```
# Convertir una cadena de valores en una matriz de datos.
def create_dataset(dataset, look_back=1):
        dataX, dataY = [], []
        for i in range(len(dataset)-look_back-1):
                a = dataset[i:(i+look_back), 0]
                dataX.append(a)
                dataY.append(dataset[i + look_back, 0])
        return numpy.array(dataX), numpy.array(dataY)
# Cargar los datos.
dataframe = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
                            engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
# Normalizar el conjunto de datos.
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# Dividir los conjuntos de entrenamiento y los de prueba.
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# Remodela a X=t e Y=t+1.
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
# Crear y ajustar una red LSTM.
model = Sequential()
model.add(LSTM(4, input_dim=look_back))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, nb_epoch=100, batch_size=1, verbose=2)
# Realizar predicciones.
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# Invertir predicciones.
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# Calcular el RECM.
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RECM' % (trainScore))
# Predicciones para los datos de entrenamiento.
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# Predicciones para los datos de prueba.
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# Ploteo.
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

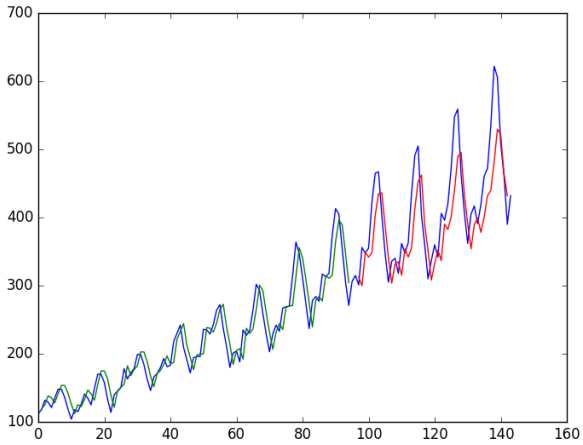


Figura 3.23: Gráfica azul: Datos reales. Gráfica verde: datos predichos sobre los datos de entrenamiento. Gráfico rojo: datos predichos sobre los datos de prueba.

CLASE MinMaxScaler(feature_range=(0,1)):
Se encuentra en la librería **sklearn.preprocessing**. Transforma caraterísticas a un escala diferente en un rango determinado, el cual por defecto es (0,1). El método usado en la aplicación es el siguiente:

- **fit_transform(X)**: Ajusta los datos **X** y luego los transforma.
- **inverse_transform(X)**: Realiza el cambio de los datos **X** a su escala original.

CLASE Sequential():
Se encuentra en la librería **keras.models**. Genera una red de capas ocultas vacías. Los métodos usados en la aplicación son los siguientes:

- **add(capas)**: Adiciona **layers**(capas) a la red. En el caso de la aplicación tomamos layers como funciones de la librería **keras.layers** como sigue:
  - **LSTM(n,input_dim=m)**: Returna **n** bloques LSTM con una dimensión de entrada **m**.
  - **Dense(n)**: Returna **n** unidades de salida.
- **compile(loss="mean_squared_error", optimizer="adam")**: Configura el proceso de aprendizaje. Sus paramétros son: **loss**, el cual es la función a minimizar; **optimizer**, un objeto optimizador.
- **fit(X, Y, nb_epoch=10, batch_size=32, verbose=1)**: Entrena el modelo para un número fijo de iteraciones. Sus parámetros son: **X**, vectores de entrada; **Y**, targets; **nb_epoch**, número de iteraciones; **batch_size**, número de muestras por actualización del gradiente; **verbose**, 0 para no registrar, 1 para registrar una barra de progreso y 2 para una línea de registro por cada iteración.

Capítulo 4

Conclusiones y Recomendaciones

Se ha logrado identificar que tipos de redes neuronales son llevadas a ciertos problemas de acuerdo a su estructura y proceso de aprendizaje, diseñando algoritmos de programación que satisfagan una respuesta. Además se ha analizado la complejidad de los procesos por cada red neuronal definida implementándose las técnicas necesarias en su desarrollo. Adicionalmente, debido a los procesos y pseudocódigos desarrollados pueden elaborarse los algoritmos en lenguajes de programación que sean de mejor uso para el lector.

Las aplicaciones en sí, son particularidades de aplicaciones mucho más amplias por lo que si englobamos nuestro problema a un escenario con mayor dificultad, se puede tomar como base lo desarrollado pues el análisis algorítmico sería el mismo. Cabe mencionar, que estas aplicaciones pueden reformularse con el objetivo de generar resultados similares sobre otro tipo de problemas.

En el presente trabajo, solo se ha tomado como base el aprendizaje supervisado y el no supervisado, por lo que es necesario tener en cuenta que existiéndo varías técnicas de aprendizaje en Machine Learning, pueden elaborarse muchas más redes neuronales artificiales usando estas técnicas y diseñando pseudocódigos y algoritmos de programación respetivos.

Siendo la red LSTM una de las más complejas en su desarrollo, es fundamental la investigación en este campo sobre redes más complejas que no han sido vistas en este trabajo, las cuales con uso de las técnicas de aprendizaje y técnicas IVS vistas se puede lograr sistematizar un proceso.

Bibliografía

[1] Suzuki, K. (2013). *Artificial Neural Networks: Architectures and Applications*. Intech Publishers.

[2] Suzuki, K. (2011). *Artificial Neural Networks: Methodological advances and Biomedical Applications*. Intech Publishers.

[3] Basogain, X. (2008). *Redes Neuronales Artificiales y sus aplicaciones*. University of Cincinnati, United States.

[4] Leondes, C. (2003). *Intelligent Systems: Technology and Applications*. Publisher: CRC Press.

[5] Wakuya, H.; Shida, K. (2003). *Systems and Computers in Japan*. Publisher: Wiley Periodicals, Inc. Syst Comp Jpn.

[6] Bullinaria, J. (2015). *Radial Basis Function Networks: Algorithms*. University of Birmingham, United Kingdom.

[7] Bullinaria, J. (2004). *Self Organizing Maps: Fundamentals*. University of Birmingham, United Kingdom.

[8] Bullinaria, J. (2015). *Learning in Multi-Layer Perceptrons - Back-Propagation*. University of Birmingham, United Kingdom.

[9] Popoviciu, N.; Boncut, M. (2005). *On the Hopfield algorithm - Foundations and examples*. Hyperion University of Bucharest and University of Sibiu, Romania.

[10] Hochreiter, S.; Schmidhuber, J. (1997). *Long Short-Term Memory*. Germany: Faculty of Informatics, Technical University of Münich. Switzerland: IDSIA Corso Elvezia 36 6900 Lugano.

[11] Veloso, M. (2001). *Perceptrons and Neural Networks*. Carnegie Mellon University, United States.

[12] Pham, D.; Liu, X. (2007). *International Journal of Systems Science: Training of Elman networks and dynamic system modelling*. Published online.