

Juan Vorobioff
Santiago Cerrotta
Nicolas Eneas Morel
Ariel Amadio



Inteligencia Artificial

y

Redes Neuronales

Fundamentos, Ejercicios

y

Aplicaciones con Python y Matlab



UTN.BA

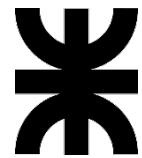
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

CiN REUN

Red de Editoriales
de Universidades Nacionales
de la Argentina

edUTecNe





Inteligencia Artificial y Redes Neuronales: Fundamentos, Ejercicios y Aplicaciones con Python y Matlab

Juan Vorobioff

Santiago Cerrotta

Nicolas Eneas Morel

Ariel Amadio

Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

2022

Inteligencia Artificial y Redes Neuronales fundamentos, ejercicios y aplicaciones con Python y Matlab / Juan Vorobioff ... [et al.] ; editado por Fernando Cejas. - 1a ed. - Ciudad Autónoma de Buenos Aires : edUTecNe, 2022.

Libro digital, PDF

Archivo Digital: descarga y online

ISBN 978-987-4998-82-8

1. Inteligencia Artificial. I. Vorobioff, Juan. II. Cejas, Fernando, ed.

CDD 006.3



Universidad Tecnológica Nacional – República Argentina

Rector: Ing. Rubén Soro

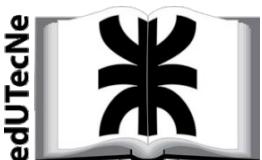
Vicerrector: Ing. Haroldo Avetta



Universidad Tecnológica Nacional – Facultad Regional Buenos Aires

Decano: Ing. Guillermo Oliveto

Vicedecano: Ing. Andrés Bursztyn



edUTecNe – Editorial de la Universidad Tecnológica Nacional

Coordinador General a cargo: Fernando H. Cejas

Queda hecho el depósito que marca la Ley Nº 11.723

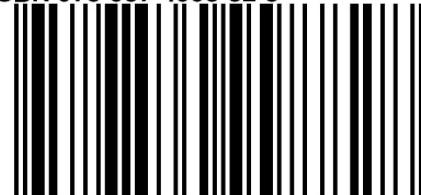
© edUTecNe, 2022

Sarmiento 440, Piso 6 (C1041AAJ)

Buenos Aires, República Argentina

Publicado Argentina – Published in Argentina

ISBN 978-987-4998-82-8



9 789874 998828



Reservados todos los derechos. No se permite la reproducción total o parcial de esta obra, ni su incorporación a un sistema informático, ni su transmisión en cualquier forma o por cualquier medio (electrónico, mecánico, fotocopia, grabación u otros) sin autorización previa y por escrito de los titulares del copyright. La infracción de dichos derechos puede constituir un delito contra la propiedad intelectual.

A nuestras familias.

Agradecimientos

Depto. de Micro y Nanotecnología - Comisión Nacional de Energía Atómica (CNEA)

Facultad Regional Buenos Aires - Universidad Tecnológica Nacional (FRBA – UTN)

Facultad Regional Delta - Universidad Tecnológica Nacional (FRD – UTN)

Facultad Regional General Pacheco - Universidad Tecnológica Nacional (FRGP – UTN)

Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Introducción a Redes Neuronales

**Algoritmos de entrenamiento de redes: propagación hacia atrás
y otros algoritmos**

Redes de funciones de base radial (RBF)

**Reconocimiento estadístico de patrones, aprendizaje
automático y redes neuronales**

Redes Neuronales con Mapas Autoorganizados (SOM)

Redes Neuronales Dinámicas

Redes Neuronales Convolucionales

Inteligencia Artificial en Dispositivos Móviles y de IoT

Visión Artificial

Aplicaciones con métodos combinados de Inteligencia Artificial

**Aplicaciones. Procesado de señales interferométricas con redes
neuronales: Estimando frecuencias**

**Redes Neuronales aplicadas a la detección de picos de glucosa
después de las comidas en series temporales de pacientes
diabéticos tipo II**

Lista de ejercicios

Epílogo

Contenidos

Prólogo

Estimado lector:

Este libro se basa en la experiencia profesional de los autores en el área de Inteligencia Artificial (IA) y redes neuronales aplicado al análisis y procesamiento de señales e imágenes.

La inteligencia artificial creció notablemente en las últimas décadas, simula procesos de aprendizaje que realizan los humanos mediante algoritmos y máquinas flexibles. Forma parte del cambio tecnológico que estamos viviendo y tiene efectos en la vida cotidiana. Se puede observar esto en la proliferación de programas inteligentes en celulares, buscadores de internet y diferentes aplicaciones de software que en cierta forma estudian el comportamiento de las personas. La IA también se utiliza en muchas áreas de las ciencias y de la ingeniería, por ejemplo, en física, química, economía, ciencias de la tierra, automatización, ciencias sociales, biología, medicina, etc. Esto puede observarse en innumerables publicaciones de artículos de distintas disciplinas.

La Inteligencia artificial no es solo que la computadora resuelva aplicaciones en forma automática, sino que debe aprender con el transcurso del tiempo y también debe mejorar sus algoritmos. Es necesario que mejore las respuestas para detectar detalles más sutiles y adaptarse a los cambios del entorno.

Más allá de que la IA imite ciertos comportamientos humanos, los algoritmos que lo componen se deben analizar e implementar correctamente para obtener buenos resultados. Así también los resultados de las maquinas no siempre son suficientes. Por ejemplo, en el procesamiento de imágenes médicas es indispensable que participen las personas. Esto nos lleva a pensar:

La tecnología debe estar acompañada de personas.

La IA contiene distintas disciplinas como aprendizaje automático, aprendizaje profundo, minería de datos, entre otras. También se puede separar en métodos estadísticos y redes neuronales. En este libro analizamos principalmente las redes neuronales. Sin embargo, también presentamos algunos métodos de reconocimiento estadístico de patrones (REP) y métodos de visión artificial, ya que presentan algunas ventajas (y desventajas) respecto de las redes neuronales. Dentro de las redes neuronales presentamos una introducción a las redes básicas, a las redes avanzadas convolucionales y a las redes dinámicas recurrentes y no recurrentes.

Existen muchas técnicas y herramientas para el procesamiento de datos. En este libro se presentan algunos métodos mediante un marco teórico, desarrollos matemáticos, ejercicios prácticos analíticos, ejemplos de distintas aplicaciones y ejercicios en Matlab®, Simulink y Python. También se agregan ejemplos en Java, RapidMiner y Orange.

Se puede armar un modelo de procesamiento de datos, o una red neuronal simple o avanzada sin conocimientos de programación, en este libro se presentan algunas herramientas gráficas para este propósito. Sin embargo, es indispensable conocer los

fundamentos de cada modelo y de cada tipo de red neuronal para procesar adecuadamente los datos, obtener una buena generalización pasada la etapa de entrenamiento y por lo tanto permitir que los sistemas aprendan correctamente.

En los capítulos I y II se realiza una introducción a las redes neuronales y a los algoritmos de propagación hacia atrás. En el capítulo III se explican las redes RBF y en el capítulo V las redes SOM, donde se analizan las diferencias con las redes anteriores. El capítulo IV corresponde al reconocimiento de patrones con redes neuronales y con métodos estadísticos, y se explican diferentes métodos y criterios para analizar el desempeño de los algoritmos. En el capítulo VI se explican las redes neuronales dinámicas recurrentes y no recurrentes. Los capítulos VII y VIII explican diferentes técnicas de aprendizaje profundo con redes neuronales convolucionales. Se muestran ejemplos para computadoras de escritorio (Windows y Linux) y para dispositivos móviles (Android, iOS y Raspberry pi). Cabe destacar que las aplicaciones mostradas son multiplataformas, con compilación individual.

El capítulo IX se realiza una introducción a la visión artificial y se muestran algunos ejemplos. En el capítulo X se muestra un ejemplo de aplicaciones de Espectroscopia de plasma inducido por láser (LIBS) y se comparan diferentes técnicas de inteligencia artificial. En el capítulo XI se muestra una aplicación avanzada de procesado de señales interferométricas con redes neuronales: Estimando frecuencias. En el capítulo XII se muestra una aplicación muy interesante de redes neuronales para la detección de gluconeogénesis alterada y picos de glucosa. Por último, en el anexo I se presenta una lista detallada con más de 70 ejercicios incluidos en este libro.

Se espera que el contenido del libro le sirva al lector para comprender diferentes técnicas de IA, así también para desarrollar y comparar distintos algoritmos con aplicaciones científico-tecnológicas y/o comerciales.

Dr. Ing. Juan Vorobioff

Capítulo I - Introducción a Redes Neuronales

En este capítulo se realiza una introducción a la inteligencia artificial (IA). Se explican distintas ramas de la IA: aprendizaje automático y aprendizaje profundo. Se detalla la neurona simple, las funciones de activación y las topologías de las redes. Se describe una breve historia de las redes neuronales y su evolución. Se explican las primeras redes neuronales estáticas de una capa y multicapa con las ecuaciones de propagación de las entradas para obtener las salidas de la red. Estas redes son las más básicas denominadas Perceptrón, Adaline y Madaline, se detallan los procesos de aprendizaje de cada red. Resulta imprescindible comprender el funcionamiento de estas redes, para luego comprender redes más avanzadas. Así también las redes descriptas se siguen utilizando ampliamente ya que tienen baja carga computacional y brindan soluciones óptimas en numerosas aplicaciones. Se muestran ejemplos analíticos y Matlab con conjuntos de datos simples, regiones de decisión y ejemplo de compuerta AND.

Introducción a la inteligencia artificial

La inteligencia artificial (IA) creció notablemente las últimas décadas, también es conocida como inteligencia de máquina o computadora. Es una rama de la informática que mediante un software determinado tiene como objetivo analizar el entorno para tomar decisiones e imitar algunos comportamientos humanos. Utiliza reglas predeterminadas con parámetros ajustables y algoritmos de búsqueda y/o modelos de aprendizaje.

La IA también se puede describir como una máquina controlada por software para realizar tareas que realizan los humanos. Se crean sistemas inteligentes con algoritmos que son capaces de aprender del entorno y/o de los datos recibidos (Beale, 2020).

En la IA tenemos asociadas distintas ramas:

Aprendizaje automático (Machine Learning: ML)

Aprendizaje profundo (Deep Learning)

Base de datos conocimiento (Data base knowledge)

Minería de datos (Data mining)

La IA tiene dos subconjuntos principales: aprendizaje automático y aprendizaje profundo, ver Fig. 1. El aprendizaje automático (ML: Machine Learning en inglés) es una rama de la inteligencia artificial. Se utilizan algoritmos con reglas matemáticas, estos permiten a las computadoras aprender imitando en cierta manera la forma de aprendizaje de los humanos. Existen muchas técnicas como por ejemplo árboles de decisión o diferentes clasificadores. Mediante ML una máquina aprende automáticamente de datos pasados y ajusta sus parámetros sin programar explícitamente. El objetivo de ML es permitir que las máquinas adquieran conocimientos y aprendan de los datos para brindar resultados precisos (Demuth, 2018).

El aprendizaje profundo modela abstracciones de muestras u objetos a alto nivel. Se trata de una rama del aprendizaje automático donde generalmente se utilizan redes neuronales sofisticadas.

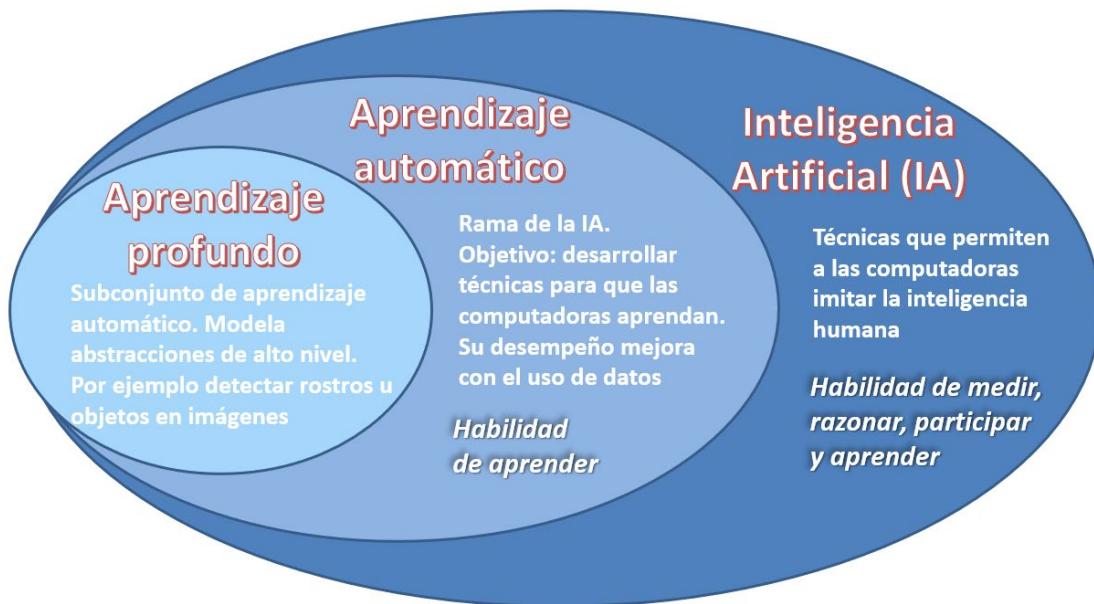


Fig. 1 – Ramas de la Inteligencia artificial

En la Tabla I mostramos las diferencias principales entre la inteligencia artificial y el aprendizaje automático.

Tabla I – Comparación entre inteligencia artificial (IA) y aprendizaje automático (ML)

Inteligencia artificial (IA)	Aprendizaje automático (ML)
La IA tiene un alcance bastante amplio	ML tiene un alcance limitado.
En la IA se trabaja para obtener un sistema inteligente capaz de realizar varias tareas sofisticadas.	ML solo realiza aquellas tareas para las que está capacitado.
El sistema de IA se encarga de maximizar las posibilidades de éxito.	El aprendizaje automático se preocupa principalmente en obtener resultados con alta precisión.
Incluye aprendizaje, razonamiento y autocorrección.	Incluye aprendizaje y autocorrección cuando se presentan nuevos datos.
La IA puede trabajar con datos no estructurados, estructurados y semiestructurados.	ML trabaja con datos estructurados y semiestructurados.

Introducción a Redes Neuronales

El cerebro está compuesto por aproximadamente 10 mil millones de neuronas, cada una de ellas conectada a aproximadamente 12 mil neuronas. Cada neurona recibe entradas electroquímicas de otras neuronas en las dendritas. Si la suma de estas entradas eléctricas es lo suficientemente potente como para activar la neurona, transmite una señal electroquímica a lo largo del axón.

Los sistemas neuronales artificiales (ANS) en cierta forma imitan la estructura del sistema nervioso, generando sistemas de procesamiento paralelos, distribuidos y adaptativos, que llegan a presentar cierto comportamiento “inteligente”. La computadora y el cerebro resultan mucho más diferentes de lo que se supone cuando se analizan algoritmos y comportamientos del cerebro. El elemento esencial de los ANS lo constituye la neurona artificial, organizada en distintas capas que forman una red neuronal. Una red neuronal, o varias interconectadas, pueden formar un sistema global de procesamiento.

En la Fig. 2 se muestra un modelo básico de una neurona artificial, y se indica su analogía con la neurona biológica. Consiste en las siguientes partes:

- Un conjunto pesos sinápticos w_{ij} correspondiente cada una de las entradas $x_j(t)$
- Una regla de propagación, por ejemplo: $h_i(t) = \sum w_{ij}x_j$ (la más común).
- Una función de activación f_i , para calcular la salida de la neurona: $y_i(t) = f_i(h_i(t))$

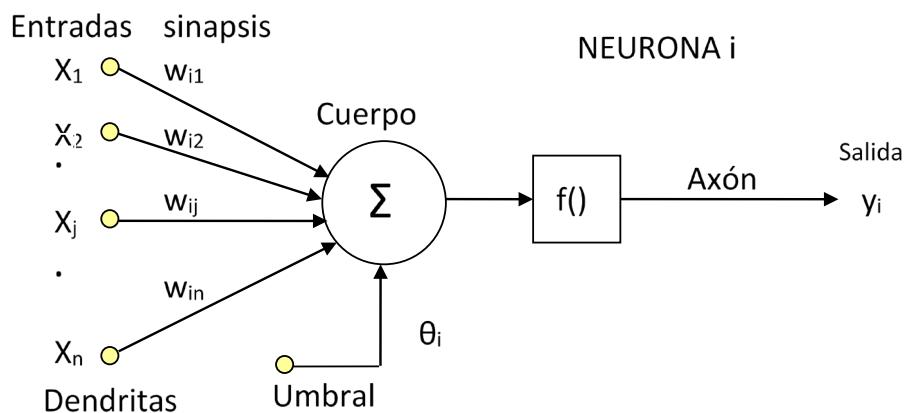


Fig. 2 – Modelo de neurona estándar

Una red neuronal artificial (RNA), también conocida como red neuronal está formada por la interconexión de muchas unidades de procesamiento llamadas neuronas, formadas por bloques no lineales distribuidas en toda la red neuronal. Las redes neuronales desde el inicio están motivadas por la forma en que procesa la información el cerebro humano, de ahí el nombre. Se puede realizar una analogía con la neurona biológica compuesta por soma, dendritas, y el axón. Las dendritas reciben los impulsos nerviosos de otras neuronas, los impulsos se procesan en el soma para ser transmitidas por medio del axón a otras neuronas (Del Brío, 2007).

La arquitectura de la red neuronal es la estructura de capas y conexiones que tiene la red. En el cerebro las neuronas se conectan por a través de sinapsis, estas conexiones son unidireccionales. Generalmente las neuronas se agrupan en capas, una o más capas constituyen la red neuronal.

En las redes neuronales se distinguen tres capas: de salida, oculta y de entrada. La capa de entrada, también llamada sensorial, está formada por neuronas que se encargan de recibir datos o señales del entorno (por ejemplo: sensores o realimentación de las salidas). La capa de salida contiene que neuronas proporcionan la salida o respuesta de la red. La capa oculta o intermedia no tiene ninguna conexión directa con el entorno. Esta capa aumenta los grados de libertad de la red, brindando mayor riqueza computacional (Hagan, 2014).

Las redes neuronales se construyen interconectando neuronas, en la Fig. 3 se muestra el bloque principal de una neurona de entrada única.

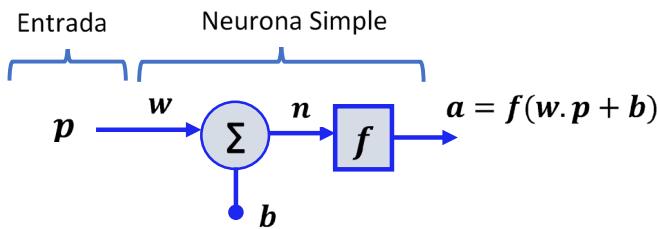


Fig. 3 – Esquema de una neurona simple

p: entrada escalar

w: peso escalar

f: función de transferencia

n: entrada neta, $n = w \cdot p + b$

La salida está dada por la siguiente ecuación:

$$a = f(n) = f(w \cdot p + b) \quad (1.1)$$

Para una red neuronal de 1 entrada, planteamos el siguiente ejemplo:

$w = 4, p = 2$ y $b = -1,5$, entonces

$$a = f(w \cdot p + b) = f(4 \cdot (2) - 1,5) \quad ; \quad a = f(6,5)$$

Existen tres operaciones funcionales distintas que tienen lugar en esta neurona de ejemplo (The Mathworks, 2019). Primero, la entrada escalar p se multiplica por el peso escalar w para formar el producto $w \cdot p$, nuevamente un escalar. En segundo lugar, la entrada ponderada $w \cdot p$ se suma al sesgo escalar b (bias) para formar la entrada neta n . El sesgo es muy parecido a un peso, excepto que tiene una entrada constante. Finalmente, la entrada neta se pasa a través de la función de transferencia f , que produce la salida escalar a . Los nombres dados a estos tres procesos son:

- función de peso, generalmente $w \cdot p$
- función de entrada neta, generalmente es la suma de las entradas ponderadas
- función de transferencia.

Para muchas clases de redes neuronales, la función de peso es un producto de un peso por la entrada, pero a veces se utilizan otras funciones de peso (por ejemplo, la distancia entre el peso y la entrada, $|w - p|$). En Matlab® se puede obtener una lista de funciones de ponderación, escribiendo `help nnweight`, se obtiene:

- `scalprod`: producto escalar.
- `normprod`: producto escalar normalizado.
- `convwf`: función de convolución.
- `negdist`: función negativa.

Se pueden utilizar las siguientes distancias:

- `boxdist` - Función de distancia de caja.

- dist - Distancia euclíadiana.
- linkdist: función de distancia de enlace.
- mandist - Función de distancia Manhattan.

La función de entrada neta más común es la suma de las entradas ponderadas con el sesgo, pero se pueden usar otras operaciones, como la multiplicación. En las redes neuronales de base radial muchas veces se suele utilizar la distancia como función de peso y también se puede usar la multiplicación como función de entrada neta. Para obtener una lista de funciones de entrada de red en Matlab®, escriba `help nnnetinput`, se obtiene:

- netprod: Función producto de entrada.
- netsum: Función suma de entrada.

Hay muchos tipos de funciones de transferencia. En Matlab®, para obtener una lista de funciones de transferencia, escriba `help nntransfer`, se muestran algunos resultados:

- hardlim: función limitador fuerte.
- logsig: función logarítmica sigmoide.
- purelin: - función lineal.
- radbas: función de base radial.
- tansig: función sigmoide simétrica.

En la Fig. 4, a la izquierda se muestra la función de transferencia limitador fuerte (en inglés *hard limit*) con la salida a en función de n . A la derecha se muestra la respuesta para neurona con 1 entrada, se gráfica la salida a en función de la entrada p . La salida de la neurona es cero si la entrada es menor a cero, y la salida es uno si la entrada es mayor o igual a cero. Se utiliza esta neurona para clasificar entradas en 2 categorías distintas. En Matlab® se implementa con la función `hardlim`. Estas transferencias se utilizan en las redes neuronales Perceptron (Hagan, 2014).

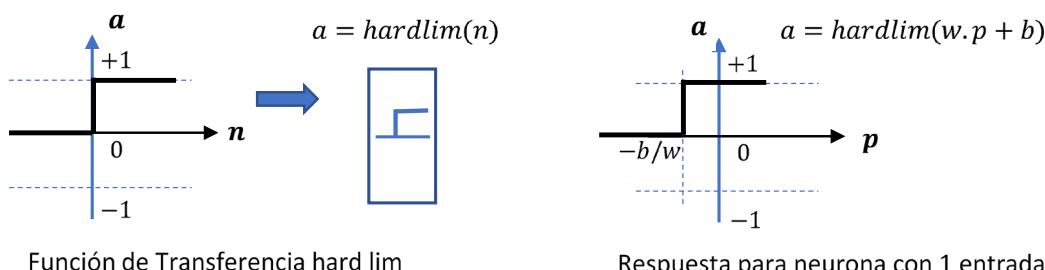


Fig. 4 – Función de transferencia limitador fuerte (*hard lim*)

En la Fig. 5, a la izquierda se observa la función de transferencia lineal con la salida a en función de n . A la derecha se muestra la respuesta para neurona con 1 entrada, se gráfica la salida a en función de la entrada p . La salida es igual a la entrada, entonces $a = n$. En Matlab® se implementa con la función `purelin`. Las neuronas con este tipo de transferencia se utilizan en las redes neuronales Adaline

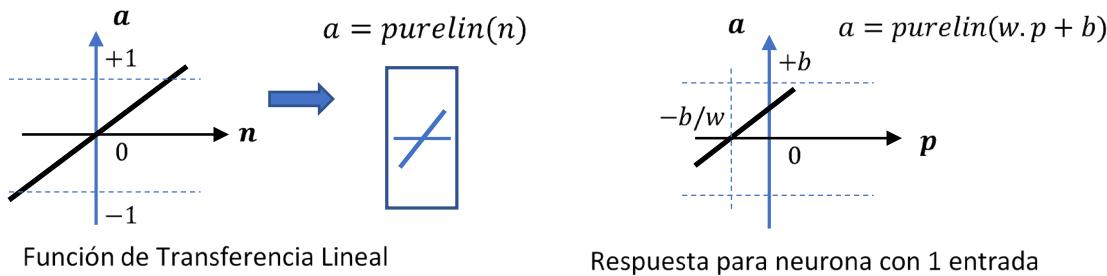


Fig. 5 – Funciones de transferencia lineal

En la Fig. 6, a la izquierda se muestra la función de transferencia sigmoide logarítmica con la salida a en función de n . A la derecha se muestra la respuesta para neurona con 1 entrada, se grafica la salida a en función de la entrada p . En Matlab® se implementa con la función *logsig*. Esta función de transferencia toma la entrada que puede valer entre menos infinito y más infinito y calcula una salida acotada entre 0 y 1 mediante la siguiente ecuación:

$$a = \frac{1}{1 + e^{-n}} \quad (1.2)$$

La función logarítmica sigmoide generalmente se utiliza en redes multicapa que se entrenan con algoritmos de propagación hacia atrás, debido a que esta función es diferenciable.

En Matlab se puede experimentar una neurona simple mediante una interfaz gráfica, escribiendo el siguiente comando: *nnd2n1*.

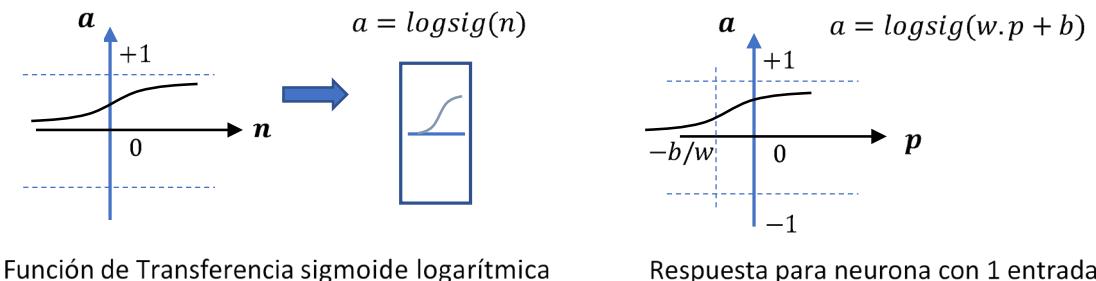


Fig. 6 – Función de transferencia sigmoide logarítmica

En la Tabla II se muestran las funciones de transferencia más utilizadas

Tabla II – Funciones de transferencia

Nombre	Relación entre entrada y salida	Repuesta	Función en Matlab®
Hard Limit	$a = 0 \text{ si } n < 0$		
Limitador Fuerte	$a = 1 \text{ si } n \geq 0$		hardlim
Limitador Fuerte simétrico	$a = -1 \text{ si } n < 0$ $a = 1 \text{ si } n \geq 0$		hardlims
Lineal	$a = n$		purelin
Lineal saturada	$a = -1 \text{ si } n < -1$ $a = n \text{ si } -1 \leq n \leq 1$ $a = 1 \text{ si } n \geq 1$		satlin
Sigmoide logarítmica	$a = \frac{1}{1 + e^{-n}}$		logsig
Sigmoide tangente hiperbólica	$a = \frac{e^n - e^{-n}}{e^n + e^{-n}}$		tansig
Competitiva	$a = 1 \text{ para neuronas con } n \text{ máximo}$ $a = 0 \text{ otros casos}$		compet

Se debe tener en cuenta que w y b son parámetros ajustables de las neuronas. Las redes neuronales resultan flexibles ya que dichos parámetros se pueden ajustar para que la red exhiba algún comportamiento deseado o interesante. Por lo tanto, puede entrenar a la red para que realice un trabajo en particular ajustando los parámetros de peso o sesgo.

Todas las neuronas del software “Neural Network Toolbox de Matlab®” se pueden usar con sesgo, se asume existente en la mayoría de las redes. Se puede omitir el sesgo en algunas neuronas.

Entradas y capas

Generalmente las redes neuronales tienen muchas entradas. Para una neurona con entradas múltiples se utiliza notación matricial para los pesos \mathbf{W} . En la Fig. 7 se muestra una neurona con R entradas. Para cada entrada individual p_1, p_2, \dots, p_R le corresponde los siguientes pesos $w_{1,1}, w_{1,2}, \dots, w_{1,R}$ de la matriz de pesos \mathbf{W} . La salida n se puede expresar con la siguiente ecuación:

$$n = w_{1,1} \cdot p_1 + w_{1,2} \cdot p_2 + \dots + w_{1,R} \cdot p_R + b \quad (1.3)$$

En forma matricial se puede escribir mediante la siguiente ecuación:

$$n = W \cdot p + b \quad (1.4)$$

La salida de la neurona denominada a se puede expresar de la siguiente forma:

$$a = f(W \cdot p + b) \quad (1.5)$$

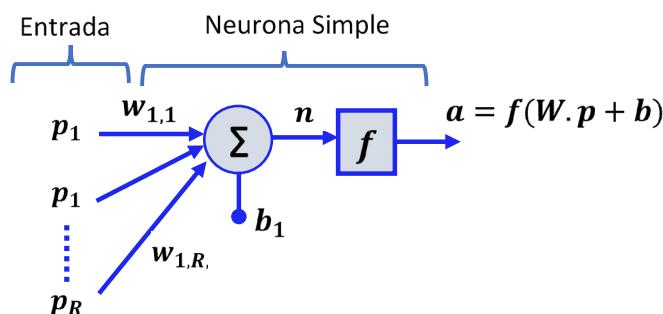


Fig. 7 – Neurona con múltiples entradas

Utilizaremos notación ampliada para describir redes con múltiples capas. En el caso de redes de múltiples capas de neuronas usamos las letras IW para matrices de pesos que están conectadas a entradas y LW para matrices de pesos que están conectadas entre capas. Es decir, se utilizan distintos nombres según sea capa intermedia o capa de entrada. También se identifica el origen y el destino de las matrices de peso.

Las matrices de peso conectadas a las entradas las llamamos matrices de ponderación o *matriz de pesos de entrada*; llamamos *matrices de pesos* a las que utilicen como entrada la salida de proveniente de otra capa. Así también, los superíndices identifican la fuente (segundo índice) y el destino (primer índice) para identificar los pesos y otros componentes de la red. En la Fig. 8 se muestra una red neuronal de entrada múltiple compuesta por una capa, utilizamos notación abreviada. Siendo S y R :

S : cantidad de neuronas de la capa 1

R : número de elementos del vector de entrada

En la figura Fig. 8 se observa que para la matriz de ponderación o pesos conectada al vector de entrada p se le asigna $IW^{1,1}$ que tiene como fuente 1 (ver segundo índice) y destino también 1 (ver primer índice). Los sesgos, entradas netas y salidas de la capa 1, tienen un superíndice 1 que indica la primera capa.

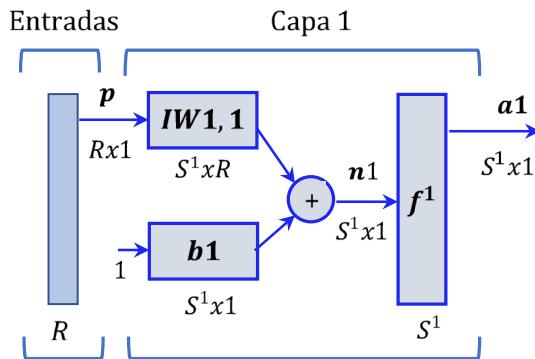


Fig. 8 – Red Neuronal con una capa de neurona

Breve historia de las redes neuronales.

En 1943, el neurofisiólogo W. McCulloch junto con el matemático Walter Pitts presentaron un artículo sobre cómo podrían funcionar las neuronas del cerebro. Modelaron una red neuronal sencilla mediante un diagrama basado en circuitos eléctricos (Del Brío, 2007).

En el año 1949, Donald Hebb escribió un trabajo que mostraba el hecho de que las vías y las conexiones neuronales se fortalecen cada vez que se utilizan, un concepto fundamentalmente esencial para las formas en que los humanos aprenden.

En la década de 1950 avanzaron mucho las computadoras, se pudo simular una red neuronal hipotética. El primer paso lo dio Rochester en los laboratorios de IBM. Este primer intento no funcionó bien. En 1959, el profesor Bernard Widrow y su alumno Marcian Hoff de la universidad de Stanford diseñaron los modelos llamados "Adaline" y "Madaline". Adaline se desarrolló para reconocer muestras binarias de modo que, si se estaba leyendo datos con bits de transmisión provenientes de una línea telefónica, podría predecir el siguiente bit de datos. La red neuronal llamada Madaline fue la primera red con una aplicación real para eliminar los ecos generados en las líneas telefónicas, utilizaba filtros adaptativos. Si bien el sistema es muy antiguo, todavía se utiliza comercialmente.

En 1962, Widrow y Hoff desarrollaron un procedimiento básico de aprendizaje que evalúa los datos antes de ajustar sus pesos, de acuerdo con la siguiente regla:

Cambio del peso $w = (\text{valor del peso previo}) * (\text{Error} / (\text{cantidad de entradas}))$.

La aplicación de esta regla genera un error si el peso previo es 0, aunque esto eventualmente se corregirá por sí solo. Si el error se conserva y se distribuye a todos los pesos, se elimina el error.

La arquitectura tradicional de von Neumann estaba ganando popularidad y desplazó temporalmente la investigación neuronal. Irónicamente, John von Neumann sugirió la imitación de redes neuronales mediante relés telegráficos o con tubos de vacío.

En el mismo período, se escribió un artículo que sugería que no podía haber una extensión de la red neuronal de una sola capa a una red neuronal de múltiples capas. Como resultado, la investigación y la financiación en redes neuronales se redujeron drásticamente. Esto se combinó con el hecho de que muchas promesas ambiciosas de las redes neuronales artificiales no se cumplieron. La idea de una computadora que se programe a sí misma es muy atractiva. Tales ideas eran atractivas pero muy difíciles de implementar.

En 1972, Kohonen y Anderson diseñaron redes similares de forma independiente entre sí. Ambos utilizaron matemáticas matriciales para describir sus ideas, pero sin darse cuenta sus propuestas eran equivalentes a circuitos Adaline analógicos.

La primera red multicapa se desarrolló en 1975, una red no supervisada. Las neuronas pueden activar un conjunto de salidas distintas.

En 1982, se renovó el interés por el campo. John Hopfield presentó un enfoque para crear máquinas más útiles utilizando líneas bidireccionales. Ya que anteriormente, las conexiones entre neuronas se realizaban solamente en una dirección. Ese mismo año, Reilly y Cooper usaron una "red híbrida" con muchas capas, cada capa usaba una estrategia diferente. Japón anunció la quinta generación sobre redes neuronales que implica inteligencia artificial. La primera generación usó interruptores y cables, la segunda generación usó el transistor, la tercera usó tecnología de estado sólido como circuitos integrados y lenguajes de programación de nivel superior, y la cuarta generación son generadores de código.

En 1986, el problema era cómo extender la regla Widrow-Hoff a múltiples capas. Tres grupos independientes de investigadores propusieron ideas similares que ahora se denominan redes de propagación hacia atrás, porque distribuye errores de reconocimiento de patrones por toda la red de adelante hacia atrás. Las redes híbridas usaban solo dos capas, estas redes de propagación hacia atrás pueden tener muchas capas.

Las redes de propagación hacia atrás pueden resultar lentas en su aprendizaje y necesitan posiblemente cientos de iteraciones para entrenarse (Kaplunovich, 2020).

En la actualidad, las redes neuronales artificiales (ANN o RNA) son muy diversas en modelos, topologías, aplicaciones, aprendizaje diseños, etc.

El uso de ANN ha crecido enormemente en los últimos años. Las redes neuronales recurrentes y las diferenciales han introducido mejoras considerables en aplicaciones industriales y científicas (Haykin, 2008).

Clasificación de Redes Neuronales

Se presentan distintas clasificaciones de las redes neuronales.

Se pueden clasificar según su aplicación o uso:

- Redes neuronales para Clasificar (clasificadores)
- Redes neuronales para Regresión (regresores)

Las redes neuronales se pueden implementar como clasificadores o como regresores. En el caso de clasificadores se asigna una clase discreta a un vector de entradas. En el caso de regresores, se asigna un vector de salida continuo o analógico a un vector de entrada continuo.

Clasificación según su topología o arquitectura

- Red neuronal monocapa – Perceptrón simple
- Red neuronal Multicapa

La red neuronal monocapa es la más sencilla, tiene una sola capa de neuronas que utilizan las entradas para obtener directamente las salidas de la red. Las redes multicapa contienen varias capas ocultas entre la capa sensorial o de entrada y la capa de salida

Clasificación según el método de aprendizaje

- Supervisado
- No supervisado
- Reforzado

En el aprendizaje supervisado, durante el entrenamiento la red dispone de los patrones de entrada y los patrones de salida deseados para esa entrada, en base a estos datos ajusta los parámetros internos de cada neurona. En cambio, en los métodos y algoritmos de aprendizaje no supervisado, no se conocen las salidas, la red neuronal ajusta sus parámetros internos en base a características comunes de los datos, es decir busca similitudes en los datos de entrada. Por ejemplo, mediante el agrupamiento calcula la distancia entre un dato y otro y forma clases según estas distancias.

En el aprendizaje reforzado solo se analiza si la respuesta es correcta o incorrecta.

Para el aprendizaje no supervisado existen distintas técnicas, por ejemplo:

- Agrupamiento
- Análisis de Componentes Principales (PCA)
- Aprendizaje competitivo.
- Mapas autoorganizados (SOM)

En el aprendizaje competitivo, las neuronas compiten entre sí. Las neuronas ganadoras son aquellas cuyos pesos se asemejan más al patrón de entrada. El aprendizaje refuerza las conexiones de la neurona ganadora y debilitar las otras.

En los Mapas autoorganizados (SOM) se agrupan los datos por similitud, para proyectar los mismos sobre un mapa y poder crear distintos grupos o clases. En el Análisis de Componentes Principales, conocido como PCA, se reduce la dimensión de los datos, describe los datos en un conjunto nuevo reducido de variables. Se busca que estas variables nuevas no estén correlacionadas. Los datos reducidos pueden ingresar a la red neuronal para reducir su complejidad. También se puede utilizar PCA en redes neuronales en forma competitiva para agrupar los datos.

Clasificación según su dependencia temporal

- Redes neuronales estáticas (dependencia estática)
- Redes neuronales dinámicas (dependencia temporal)

Las redes neuronales estáticas no tienen memoria, primero se entrena y luego transforman un conjunto de entradas para obtener de salidas. Una vez entrenadas las salidas dependen solo de las entradas. Estas redes son las más utilizadas.

En cambio, las redes neuronales dinámicas permiten establecer una relación entre las salidas y las entradas y/o salidas presentes y previas. Es decir, se agrega memoria a estas redes. Los modelos se construyen con ecuaciones diferenciales o ecuaciones en diferencia para reducir errores en las salidas.

Las redes dinámicas se pueden separar en Recurrentes (evolución de recurrencia) y Diferenciales (evolución continua).

Clasificación según su conexión

- Redes neuronales prealimentadas (feedforward en inglés)
- Redes neuronales recurrentes

Una red neuronal prealimentada es una red donde las conexiones entre las neuronas no forman un ciclo. Resultan diferentes de las redes neuronales recurrentes. En la red prealimentada, la información se mueve solo hacia adelante, desde la capa de entrada, luego a la capa oculta y luego a la capa de salida. No hay ningún bucle o ciclo en estas redes.

A continuación, se nombran algunos tipos de redes.

- Perceptrón Simple
- Perceptrón Multicapa
- Adaline y Madaline
- Redes neuronales RBF (funciones de base radial)
- Redes neuronales dinámicas recurrentes (RNN)
- Redes neuronales profundas o convolucionales (CNN)
- Redes Neuronales derivativas

Widrow y su estudiante Hoff estudiaron e introdujeron la red Adaline de una capa y la regla de aprendizaje donde usaron el algoritmo LMS (conocido como Least Mean Square). Las redes lineales (Adaline) son parecidas a la red perceptrón, pero su función de transferencia es lineal en lugar de limitante. Esto permite que sus salidas tomen valores analógicos, mientras que la salida del perceptrón está limitada a 0 o 1. Las redes perceptrón, solo resuelven problemas linealmente separables. La red Madaline es una red Adaline de múltiples capas (Zed, 2015).

Red Neuronal Multicapa

Las redes neuronales generalmente tienen varias capas. Cada capa tiene una matriz de peso W , un vector de sesgo (conocido como bias) b y un vector de salida: a . Se utiliza superíndice en la variable de interés para distinguir la pertenencia a las distintas capas en las matrices de peso, los sesgos y los vectores de salida (Mishra, 2019). En la Fig. 9 se observa el uso de la notación para una red neuronal de tres capas, y en la parte inferior de la figura se muestran las ecuaciones (Hagan, 2014).

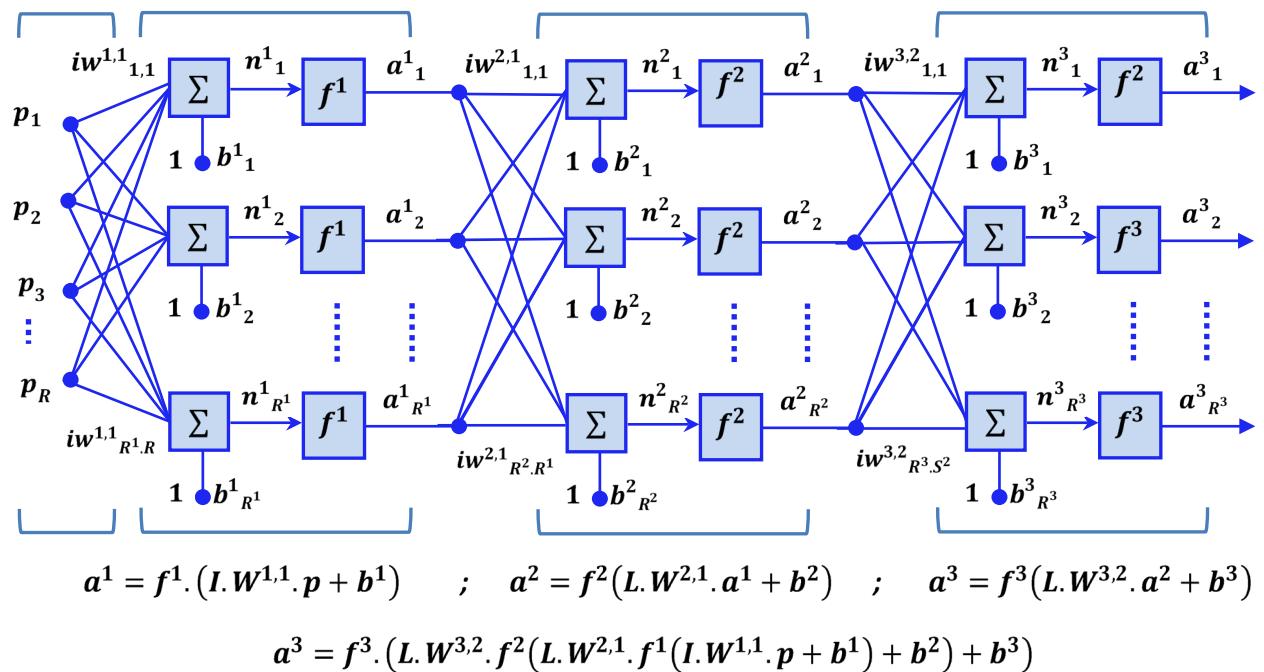


Fig. 9 – Red Neuronal Multicapa

A modo de ejemplo, observamos $LW^{3,2}$, tiene una fuente 2 (segundo índice) y un destino 3 (primer índice)

La misma red de tres capas también se puede dibujar usando notación abreviada, ver Fig. 10.

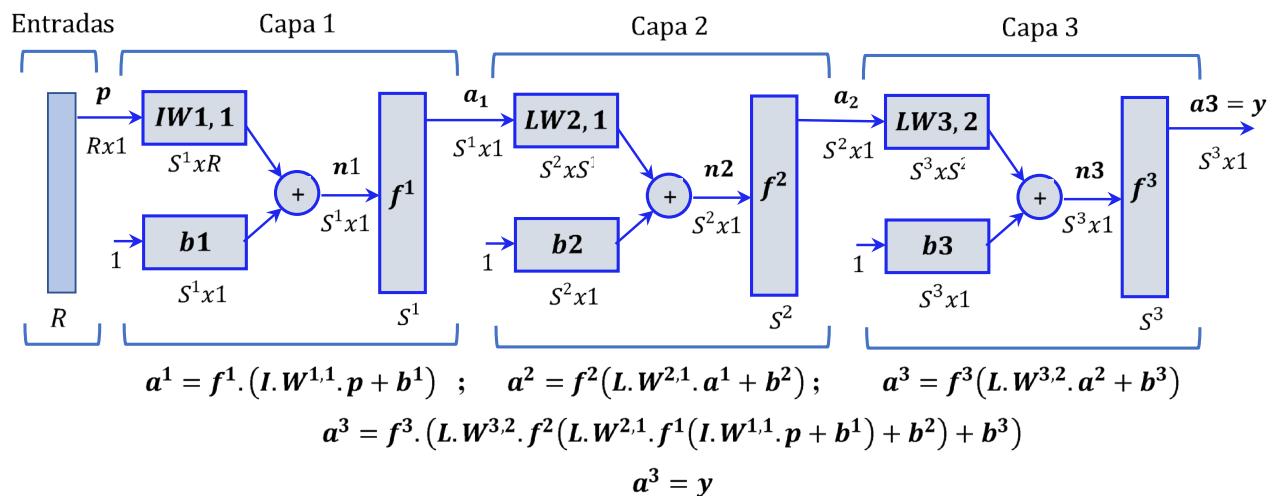


Fig. 10 – Red Neuronal Multicapa con notación abreviada

Las redes de múltiples capas resultan bastante potentes.

Mediante una red neuronal simple de dos capas, con funciones de activación sigmoideas en la primera capa y con funciones lineales en la segunda capa, se puede aproximar cualquier función (con un número finito de discontinuidades).

Esta configuración de dos capas se usa ampliamente en muchas aplicaciones de redes neuronales, juntamente con el algoritmo de propagación hacia atrás.

En el ejemplo de la Fig. 10, la salida de la red es la salida de la última capa, es decir capa 3, $y = a^3$. Donde se utiliza y para indicar la salida de la red.

Funciones de procesamiento de entradas y salidas

Las entradas de la red pueden tener funciones de procesamiento asociadas. Estas funciones transforman los datos de entrada del usuario a una forma que es más fácil o eficiente para una red.

Por ejemplo, en Matlab® la función *mapminmax* transforma los datos de entrada de modo que todos los valores caen en el intervalo $[-1, 1]$. Esto puede acelerar el aprendizaje en muchas redes. Otra función muy utilizada en Matlab® es: *removeconstantrows* elimina las filas del vector de entrada que corresponden a los elementos de entrada que siempre tienen el mismo valor, porque estos elementos de entrada no proporcionan ninguna información útil a la red. La tercera función de procesamiento común es *fixunknowns*, que se encarga de recodificar datos desconocidos (en Matlab se representan con valores NaN) en una forma numérica para la red. La función *fixunknowns* conserva información sobre qué valores se conocen y cuáles se desconocen.

De manera similar, las salidas de red también pueden tener funciones de procesamiento asociadas. Estas funciones de salidas transforman los vectores de salida para que sean iguales o similares a las salidas esperadas, por ejemplo, realizan un cambio de escala. De esta forma se producen datos de salida con las mismas características que los objetivos originales proporcionados por el usuario.

Redes neuronales en sistemas adaptativos

Para sistemas adaptativos, se trabaja con redes neuronales que aprenden del entorno, en forma supervisada. En otras palabras, obtenemos la respuesta deseada donde la red neuronal trata de ajustar sus parámetros mediante un proceso de aprendizaje iterativo. La aproximación se realiza ajustando en forma sistemática un conjunto de parámetros libres, denominados pesos sinápticos. En efecto, los pesos sinápticos proveen un mecanismo para guardar información de los datos de entrada.

En el contexto de aplicaciones de procesamiento de señales adaptativas, las redes neuronales presentan las siguientes ventajas, respecto a sistemas adaptativos lineales:

- No linealidad: se puede abarcar una gran cantidad de sistemas físicos, teniendo en cuenta que muchos sistemas tienen un comportamiento no lineal
- Capacidad de aprendizaje según las entradas y salidas del sistema en cuestión.

- Generalización: esto implica la capacidad de respuesta a sistemas desconocidos con entradas que nunca se usaron en la red
- Tolerancias a errores, teniendo en cuenta que la red neuronal completa puede tener una respuesta satisfactoria, a pesar de que algunas de sus neuronas funcionen mal.
- Integración a gran escala y capacidad de procesamiento en paralelo

Red Neuronal Perceptron

Las primeras neuronas artificiales fueron desarrolladas en 1943 por W. McCulloch y W. Pitts. En este modelo se realiza una suma ponderada de señales de entrada y se compara con un umbral o límite para obtener la salida de la neurona. Si la suma es mayor o igual al límite umbral, la salida es 1 y si la suma es menor, la salida es 0. Las redes con estas neuronas podrían calcular cualquier aritmética o función lógica. A diferencia de las redes biológicas, los parámetros de estas redes simples tuvieron que diseñarse, ya que no existía ningún método de entrenamiento. Sin embargo, se generó un gran interés y una gran inspiración en computadoras digitales inspiradas en neuronas biológicas.

En 1957, F. Rosenblatt junto con otros investigadores desarrollaron las primeras redes neuronales llamadas perceptrones. Utilizaban neuronas similares las redes de McCulloch y Pitts. Sin embargo, introdujeron una regla de aprendizaje para entrenar redes de perceptrones para poder resolver problemas de reconocimiento de patrones. Este paso fue clave en el avance de redes neuronales. Demostraron que su regla de aprendizaje siempre va a converger a los pesos correctos de la red, siempre y cuando existan pesos que resuelvan el problema. Se presentaron ejemplos de comportamiento adecuado a la red, que aprendió de sus errores. Mediante la regla de aprendizaje, el perceptrón puede aprender cuando se inicializa con valores aleatorios de pesos y sesgos. Sin embargo, estas redes de perceptrones son limitadas, no son capaces de implementar algunas funciones elementales. En la década de 1980 estas limitaciones se superaron con redes de perceptrones mejoradas de múltiples capas (multicapa) y reglas de aprendizaje asociadas.

Hoy en día, el perceptrón todavía se considera una red importante, que es una red rápida y confiable para resolver algunas aplicaciones sencillas. Además, el estudio de la red perceptrón proporciona una buena base para comprender redes más complejas. Discutiremos la red perceptrón y su regla de aprendizaje.

Una neurona real biológica emite una señal de salida solo cuando la suma total de las señales de entrada excede un cierto umbral. Este fenómeno se modela en un perceptrón calculando la suma ponderada de las entradas, representando todas las señales de entrada. Como en las redes neuronales biológicas, esta salida alimenta a otras redes perceptrones (Kundella, 2020).

La red neuronal Perceptron utiliza la función de transferencia hard lim:

$$a = \text{hardlim}(W.p + b) \quad (1.6)$$

En la Fig. 11 mostramos una red neuronal Perceptron con una neurona y 2 entradas.

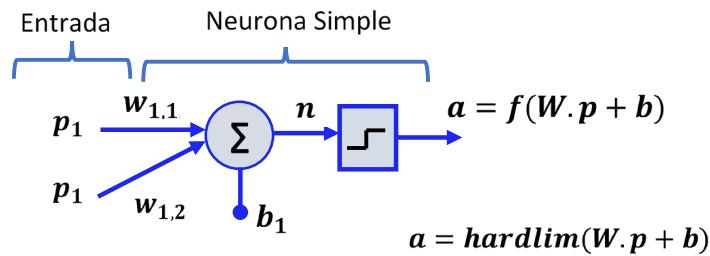


Fig. 11 – Red neuronal Perceptron simple con 1 neurona y 2 entradas

Procesos de aprendizaje de Perceptron

Para el proceso de aprendizaje supervisado se utilizan las entradas p_i y las respuestas correctas t_i , es necesario tener un grupo de entrenamiento donde se conozcan las salidas

$$\{p_1, t_1\}, \{p_2, t_2\}, \dots, \{p_n, t_n\}$$

Se utiliza a para la salida de la red y e para el error (Pytorch, 2021).

El proceso de aprendizaje del Perceptron simple se define de la siguiente manera:

- 1) Inicialización en forma aleatoria de los pesos W y los sesgos b de la red
- 2) Tomar un patrón de entrada-salida $\{p_i, t_i\}$
- 3) Calcular el valor que toma la salida de la red:

$$a_i = f(W \cdot p_i + b) \quad (1.7)$$

- 4) Se calcula el error, los pesos y los sesgos nuevos mediante las siguientes ecuaciones:

$$e_i = t_i - a_i \quad (1.8)$$

$$W^{nuevo}_i = W^{anterior}_i + e_i \cdot p_i^T \quad (1.9)$$

$$b^{nuevo}_i = b^{anterior}_i + e_i \quad (1.10)$$

Este paso también se puede realizar de la siguiente forma:

Si $a_i = t_i$ (clasificación correcta), entonces se mantienen los pesos y los sesgos

Si $a_i \neq t_i$ (clasificación incorrecta), entonces actualizamos pesos W^{nuevo}_i y sesgos b^{nuevo}_i con las ecuaciones anteriores

- 5) Se repite el paso 2 hasta completar el conjunto completo de entrenamiento.
- 6) Luego, se repiten los pasos 2, 3, 4 y 5 se repiten hasta que se alcanza el criterio de detención

Existen distintos criterios de detención del perceptrón simple:

Criterio 1: Establecer un número máximo de ciclos. Generalmente, se elige el número que proporciona el mayor porcentaje de éxito en la clasificación de patrones de entrenamiento. Este número no necesariamente es el obtenido en el último ciclo

Criterio 2: Cuando la tasa de éxito en la clasificación de patrones de entrenamiento no cambia durante cierta cantidad de ciclos

Criterio 3: Se puede utilizar un conjunto de validación, que correspondería a una parte aleatoria del conjunto de entrenamiento. En este caso, el criterio de detención es cuando el mejor porcentaje de éxito en los patrones de validación no aumenta o permanece estable a lo largo de cierta cantidad de ciclos.

Ejercicios Matlab® y Analíticos

Ejercicio 1.1

Calculo analítico de Red Neuronal Perceptron

Diseñar una red neuronal Perceptron para el siguiente grupo de entrenamiento

$$\left\{ p_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, t_1 = [0] \right\}, \quad \left\{ p_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = [1] \right\}$$

Cada vector de entrada está compuesto por 3 variables distintas. Se puede dar el siguiente ejemplo de clasificación de frutas, el primer elemento de la entrada p_i puede indicar el color, utilizando 1 si es fruta roja y -1 si es verde, el segundo elemento de p_i puede indicar el tamaño, utilizando 1 si es grande y -1 si es chico. Por último, el tercer elemento de p_i puede indicar la dureza, utilizando 1 si corresponde fruta dura y -1 si es blanda. Las salidas t_i pueden corresponder, por ejemplo 0 para una fruta y 1 para otra fruta.

Resolución

El vector de pesos W y sesgo b se inicializan en forma aleatoria, generalmente con valores bajos, por ejemplo

$$W = [0,5 \ 0 \ 1] \text{ y } b = 1$$

Se comienza la primera iteración, se aplica el vector de entrada p_1 a la red

$$a = \text{hardlim}(W \cdot p_1 + b) = \text{hardlim}\left([0,5 \ 0 \ 1] \cdot \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 1\right) = \text{hardlim}(0,5) = 1$$

$$e = t_1 - a = 0 - 1 = -1$$

$$W^{nuevo} = W^{anterior} + e.p^T = [0,5 \ 0 \ 1] - 1. [1 \ -1 \ -1]$$

$$W^{nuevo} = [-0,5 \ 1 \ 2]$$

$$b^{nuevo} = b^{anterior} + e = 1 - 1 = 0$$

Se comienza la segunda iteración con p_2

$$a = \text{hardlim}(W.p_2 + b) = \text{hardlim}\left([-0,5 \ 1 \ 2].\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + 0\right) = \text{hardlim}(-1,5) = 0$$

$$e = t_2 - a = 1 - 0 = 1$$

$$W^{nuevo} = W^{anterior} + e.p^T = [-0,5 \ 1 \ 2] + 1. [1 \ 1 \ -1] = [0,5 \ 2 \ 1]$$

$$b^{nuevo} = b^{anterior} + e = 0 + 1 = 1$$

En la tercera iteración se utiliza la entrada p_1

$$a = \text{hardlim}(W.p_1 + b) = \text{hardlim}\left([0,5 \ 2 \ 1].\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 1\right) = \text{hardlim}(-2,5) = 0$$

$$e = t_1 - a = 0 - 0 = 0$$

$$W^{nuevo} = W^{anterior} + e.p^T = [0,5 \ 2 \ 1] + 0. [1 \ -1 \ -1] = [0,5 \ 2 \ 1]$$

$$b^{nuevo} = b^{anterior} + e = 1 - 0 = 1$$

En la cuarta iteración se utiliza la entrada p_2

$$a = \text{hardlim}(W.p_2 + b) = \text{hardlim}\left([0,5 \ 2 \ 1].\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + 1\right) = \text{hardlim}(1,5) = 1$$

$$e = t_2 - a = 1 - 1 = 0$$

$$W^{nuevo} = W^{anterior} + e.p^T = W^{anterior} ; \text{No se modifica por tener } e = 0$$

$$b^{nuevo} = b^{anterior} + e = 0 + 1 = 1 ; \text{No se modifica por tener } e = 0$$

Observamos que en la tercera y cuarta iteración los errores valen cero, es decir que clasifica correctamente

Ejercicio 1.2

Regiones de decisión para de Red Neuronal Perceptron

Mediante el método gráfico conociendo los pesos y el sesgo de una red neuronal Perceptron, se pide graficar las regiones de decisión

$$w_{1,1} = 1 ; w_{1,2} = 1 ; b = -1$$

Resolución

Utilizando la configuración de la Fig. 11, la red neuronal Perceptron utiliza la función de transferencia hard lim (1.6).

La salida de la red está determinada por

$$a = \text{hardlim}(n) = \text{hardlim}(W \cdot p + b)$$

$$a = \text{hardlim}(w_{1,1} \cdot p_1 + w_{1,2} \cdot p_2 + b)$$

Los límites de decisión están determinados para los vectores de entrada donde n vale cero

$$n = w_{1,1} \cdot p_1 + w_{1,2} \cdot p_2 + b = 0$$

Tomando de ejemplo los siguientes valores:

$$w_{1,1} = 1 ; w_{1,2} = 1 ; b = -1$$

Reemplazando estos valores, obtenemos

$$n = w_{1,1} \cdot p_1 + w_{1,2} \cdot p_2 + b = p_1 + p_2 - 1 = 0$$

Esta ecuación $p_1 + p_2 - 1 = 0$ genera una recta que separa 2 regiones.

Para $p_2 = 0$, despejamos p_1 y obtenemos $p_1 = 1$

Para $p_1 = 0$, despejamos p_2 y obtenemos $p_2 = 1$

Los resultados de la recta y las 2 regiones se observan en la Fig. 12. Para saber la región donde se cumple $a = 1$, por ejemplo, podemos probar con una entrada $p = [3 \ 0]^T$

$$a = \text{hardlim}(n) = \text{hardlim}(w^T \cdot p + b) = \text{hardlim}\left([1 \ 1] \cdot \begin{bmatrix} 3 \\ 0 \end{bmatrix} - 1\right) = \text{hardlim}(2) = 1$$

Es decir que para la entrada $[3 \ 0]$ la salida de la red neuronal vale 1, de esta forma separamos 2 regiones en el gráfico, $a = 1$ y $a = 0$

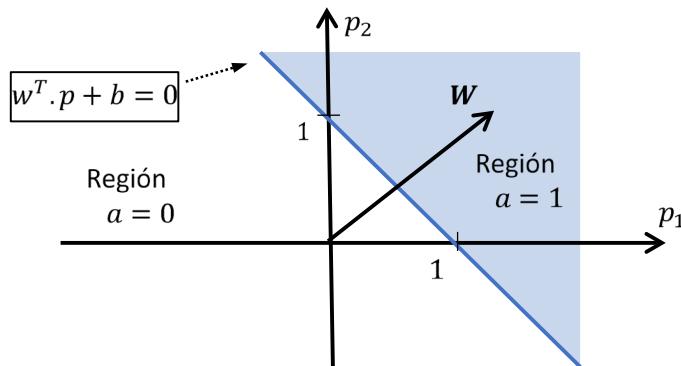


Fig. 12 – Regiones de decisión para una neurona simple Perceptron

A su vez, las regiones se pueden obtener gráficamente. Se gráfica el vector $w^T = [1 \ 1]$, luego la recta que separa las regiones siempre resulta perpendicular al vector w . Se elige un punto que cumpla $w^T \cdot p + b = 0$. Se observan los resultados con la recta de clasificación en la Fig. 12.

Teniendo en cuenta el producto $w^T \cdot p = -b$, entonces el vector de pesos w^T siempre apunta a la región de $a = 1$

Ejercicio 1.3**Cálculo de Neuronal Perceptron para compuerta AND y OR**

Dada la función lógica de compuerta AND, se representan las entradas con p_i y para las respuestas correctas se utiliza t_i

$$p_1^T = [0 \ 0], t_1 = 0$$

$$p_2^T = [0 \ 1], t_2 = 0$$

$$p_3^T = [1 \ 0], t_3 = 0$$

$$p_4^T = [1 \ 1], t_4 = 1$$

Ejemplo de red perceptrón para implementar las compuertas AND y OR. Se pide:

- a) Diseñar analíticamente la red perceptrón para compuerta AND
- b) Diseñar con Matlab® mediante función perceptrón, evaluar funcionamiento y mostrar matriz de confusión (The Mathworks, 2019)
- c) Diseñar con Matlab® mediante función propia y evaluar funcionamiento
- d) Repetir pasos anteriores para compuerta OR

Resolución

a) Se grafican los puntos con círculos vacíos para el caso de salida cero y con círculos llenos para el caso de salida 1. Se traza una recta de límites de decisión que separe los círculos vacíos de los círculos llenos, existen infinitas soluciones, se traza alguna razonable. Luego se traza un vector de pesos w que sea perpendicular a la recta anterior, se elige algún largo, por ejemplo, $w^T = [3 \ 3]$. Se observan estas operaciones en la Fig. 13.

Hace falta encontrar el valor del sesgo b , por ejemplo, se toma el punto $[1,5 \ 0]$ correspondiente a la recta de límite de decisión, debe cumplir $w^T \cdot p + b = 0$, entonces:

$$w^T \cdot p + b = [3 \ 3] \cdot \begin{bmatrix} 1,5 \\ 0 \end{bmatrix} + b = 0 ; 4,5 + b = 0 ; b = -4,5.$$

Probamos la red para el punto $p_3^T = [1 \ 0]$

$$a = \text{hardlim}(n) = \text{hardlim}(w^T \cdot p + b) = \text{hardlim}\left([3 \ 3] \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} - 4,5\right) = \text{hardlim}(-1,5) = 0$$

Verificamos que funciona correctamente en el punto p_3^T

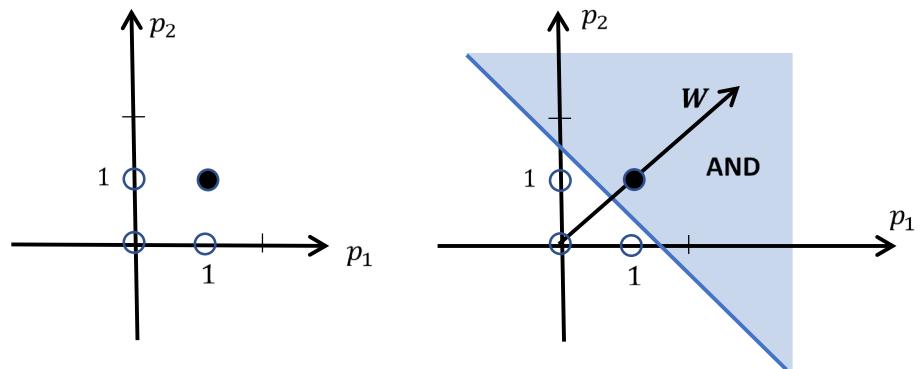


Fig. 13 – Red neuronal Perceptron para compuerta AND

b)

% Ejemplo AND Perceptron

close all; clc; clear all;

x = [0 0 1 1 ; 0 1 0 1]; % son 4 entradas: 00 01 10 11

t = [0 0 0 1]; % 4 salidas

% Inicializamos los pesos y el bias (opcional)

%net1.IW {1,1} = [3 3]; net1.b {1} = -4.5;

net1 = perceptron;

net1 = train(net1,x,t);

view(net1)

% Mostramos los pesos y el bias (opcional)

W = net1.IW {1,1}

b = net1.b {1}

% Ingresamos las entradas a la red, obtenemos salidas y, comparamos con t

y = net1(x);

errores = y -t

Resultados Matlab®:

errores =

0 0 0 0

Es decir que la red clasificó correctamente todos los datos de entrada

En la Fig. 14 se observan los resultados Matlab® de la configuración de la red. Mediante el botón de “Confusión” se pueden observar diferentes matrices de confusión, según sea entrenamiento, validación, testeo y todas las entradas. Se indica el porcentaje de clasificación correcta.



Fig. 14 – Red neuronal y Matrices de Confusión

c) Compuerta AND

```

function prueba_perceptron
clc ; clear all; close all
% Compuerta And:
P= [0 0 1 1; 0 1 0 1]; t= [0 0 0 1]; w=[0 0]; b=0; max_repetic=20;
[w b]=my_perceptron(P,t,w,b, max_repetic);
T=hard_limit(w*P+b)
error = T -t
end
function [w,b]=my_perceptron(P,t,w,b, max_repetic)
% P: Vector de entradas
% t: Vector de entrenamiento (respuestas correctas)
% w: pesos ; b: bias
% max_repetic: cantidad máxima de repeticiones
% Aprendizaje del Perceptrón, se repiten estos pasos hasta obtener
% clasificación correcta de las entradas o hasta alcanza máximo de repeticiones
% W_nuevo = W_viejo + e*p
% e   = t - a
% b   = b_viejo + e
respuesta_ok =0; cuenta=0;
[~, columnas]=size(P);
% Graficamos entradas
plotpv(P,t) ;grid on; linehandle = plotpc(w,b);
while respuesta_ok ~=8
    for i=1:columnas
        out=hard_limit(w*P(:,i)+b); % Función hard limit

```

```

respuesta_ok =respuesta_ok +1;
if respuesta_ok ==8
    break; % FIN
end
% Aprendizaje del Perceptron
if out ~=t(i)
    respuesta_ok =0;
    e=t(i)- out; % Calculamos error
    % Ajustamos peso y bias
    w=w+(e*P(:,i)'); b=b+e;
end
linehandle = plotpc(w,b,linehandle);
drawnow; pause(0.5)
end
cuenta=cuenta+1
if cuenta==max_repetic
    cuenta
    break;
end
end
function salida = hard_limit(entrada)
% salida = 0 if entrada<0
% salida = 1 caso contrario
for i=1: numel(entrada)
    if entrada(i)<0
        salida(i)=0;
    else
        salida(i)=1;
    end
end
end

```

Perceptron de múltiples neuronas

En la Fig. 12 se tiene 1 límite de decisión correspondiente a una neurona, solo se separan 2 categorías. La red Perceptron con múltiples neuronas puede clasificar las entradas en varias categorías, cada una de ellas es representada por un vector de salida distinto. Debido a que cada salida puede ser 0 o 1, para S neuronas se tiene 2^S posibles categorías. Se representan los límites de cada neurona con la siguiente ecuación, el subíndice i indica el número de neurona.

$$w^T_i \cdot p + b_i = 0 \quad (1.11)$$

En la Fig. 15 se muestra un ejemplo para separar 4 categorías, se utilizan 2 neuronas

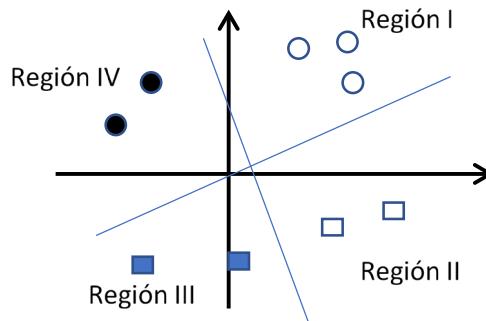


Fig. 15 – Red neuronal Perceptron para compuerta AND

Limitaciones del Perceptron

Muchas aplicaciones no se pueden separar linealmente. Por ejemplo, la red clásica Perceptron no separa correctamente la compuerta XOR. Se representan las entradas con p_i y para las respuestas correctas se utiliza t_i

$$p_1^T = [0 \ 0], t_1 = 0$$

$$p_2^T = [0 \ 1], t_2 = 1$$

$$p_3^T = [1 \ 0], t_3 = 1$$

$$p_4^T = [1 \ 1], t_4 = 0$$

Se grafican los puntos con círculos vacíos \circ para el caso de salida cero y con círculos llenos \bullet para el caso de salida 1. No se puede agregar una recta que separe los círculos llenos de los vacíos. En la Fig. 16 se muestran ejemplos de problemas linealmente inseparables

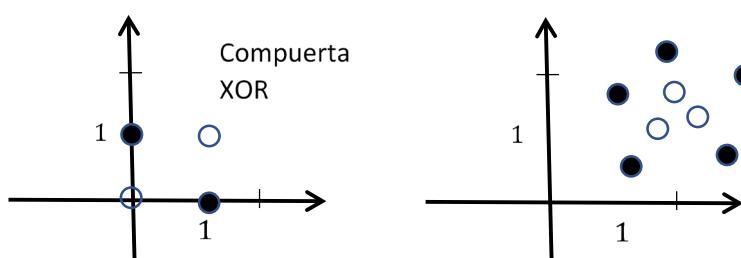


Fig. 16 – Problemas linealmente inseparables

En la década de 1980 estas limitaciones se superaron con redes de perceptrones mejoradas (multicapa) y reglas de aprendizaje asociadas.

El problema de resolver la función XOR se soluciona mediante una red neuronal con dos capas (vector de entradas, capa intermedia u oculta y capa de salida). Se utilizan dos neuronas en la capa oculta y solo una neurona en la capa final o de salida, se muestra esta red en la Fig. 17. Todas las neuronas que forman la capa oculta dividen el plano en dos regiones, como se

muestra en la Fig. 18. De esta forma funciona correctamente la red neuronal para la compuerta XOR.

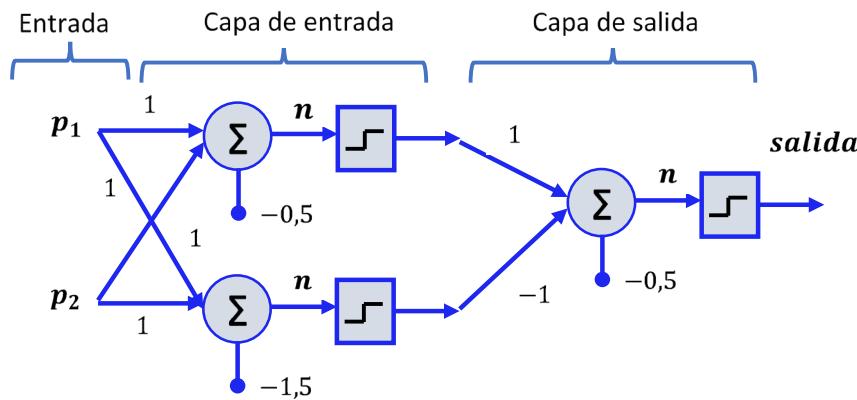


Fig. 17 – Red neuronal Perceptron para compuerta XOR

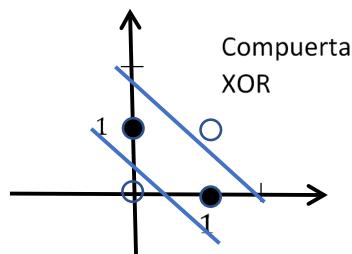


Fig. 18 – Regiones de decisión de la compuerta XOR

Filtros Adaptativos con redes neuronales MLP (Multi Layer Perceptron)

En la Fig. 19 se muestra un esquema básico de filtros adaptativos con redes neuronales

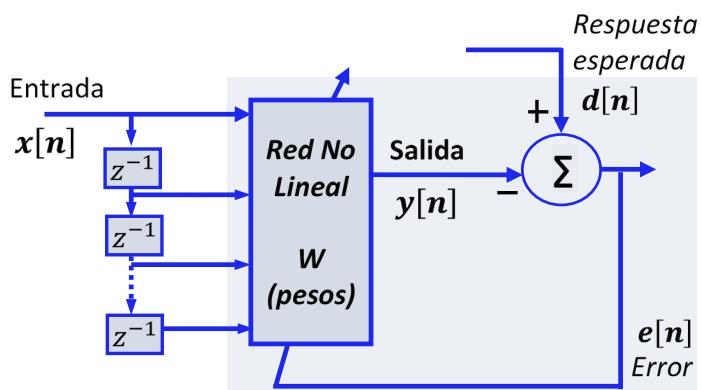


Fig. 19 –Filtro Adaptativo no lineal con elementos de retardo, basado en redes neuronales

Red neuronal Adaline y Madaline

Las redes neuronales Adaline utilizan las funciones de transferencia lineales. Tomamos de ejemplo una red Adaline con solo una neurona y dos entradas. La Fig. 20 muestra el diagrama de esta red (Demuth, 2018), (Tensorflow, 2021).

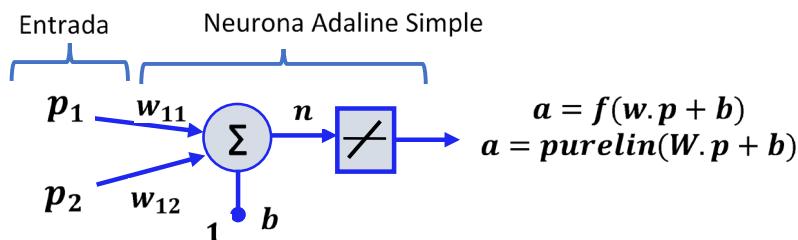


Fig. 20 – Esquema de una red neuronal simple ADALINE de 1 neurona

La matriz de pesos W en este caso tiene solo una fila. La salida de la red (a) se calcula con la función de transferencia:

$$a = \text{purelin}(n) = \text{purelin}(W \cdot p + b) = W \cdot p + b \quad (1.12)$$

Para este ejemplo:

$$a = w_{11} \cdot p_1 + w_{12} \cdot p_2 + b \quad (1.13)$$

Así como el perceptrón, el Adaline tiene un límite de decisión que está determinado por el vector de entrada para los cuales la entrada neta n es cero. Para $n = 0$ se tiene la ecuación:

$$W \cdot p + b = 0 \quad (1.14)$$

Esta ecuación especifica el límite de decisión, en la Fig. 21 se muestra una recta que separa 2 regiones según la ecuación anterior. Los vectores de las entradas en el área gris superior derecha generan a una salida mayor que 0. Los vectores de las entradas en el área blanca inferior izquierda generan a una salida menor que 0. Entonces, se puede usar Adaline para clasificar objetos en dos categorías.

Sin embargo, Adaline puede clasificar objetos solo cuando los mismos son linealmente separables. Por lo tanto, Adaline presenta la misma limitación que la red perceptrón.

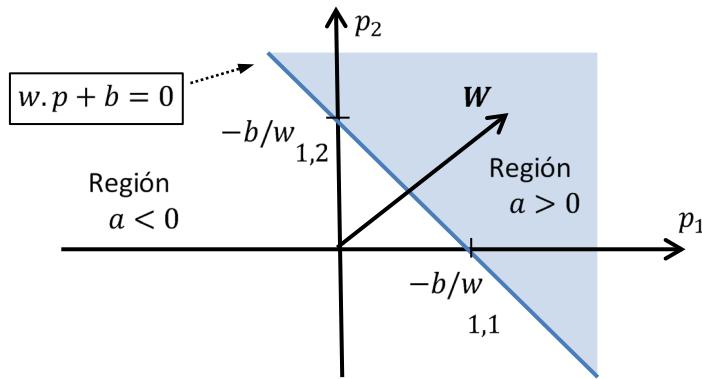


Fig. 21 – Límite de decisión para una neurona simple ADALINE

Procesos de aprendizaje Adaline

Se entrena la red para que tenga un error mínimo utilizando el algoritmo de mínimos cuadrados medios (Least Mean Squares: LMS) desarrollado por el profesor B. Widrow y su alumno T. Hoff. Se buscan los pesos óptimos de la red neuronal \$w^*\$, siendo:

$$w^* = \operatorname{argmin} J(e(w)) ; w \in \mathbb{R}^n \quad (1.15)$$

Siendo \$J(e(w))\$ el índice de rendimiento que debe ser una función convexa.

Se entrena la red neuronal para tener error mínimo mediante el algoritmo LMS (Widrow Hoff)

$$J(e(w)) = \frac{1}{2} \cdot e^T(w) \cdot e(w) = \frac{1}{2} \cdot \|e(w)\|^2 \quad (1.16)$$

Siendo \$e(w)\$ el vector de errores:

$$e(w) = [e_k]_{k=1}^N \quad (1.17)$$

$$\text{Con: } e^i = y^{*,i} - y^i \quad (1.18)$$

La salida óptima es \$Y^*(n)\$ compuesta por \$y^{*,i}

La salida real es \$Y(n)\$ compuesta por \$y^i = w^T \cdot x^i\$; siendo \$x^i\$ la entrada con \$x^i \in \mathbb{R}^m\$

Desarrollamos \$J(e(w))\$:

$$J(e(w)) = \frac{1}{2} \cdot \sum_{k=1}^N (e_k)^2 \quad (1.19)$$

$$Y(n) = [y^1, y^2, \dots, y^N] ; \quad Y(n) \in \mathbb{R}^N \quad (1.20)$$

$$Y^*(n) = [y^{*,1}, y^{*,2}, \dots, y^{*,N}] ; \quad Y(n) \in \mathbb{R}^N \quad (1.21)$$

$$X(n) = [x^1, x^2, \dots, x^N] ; \quad X(n) \in \mathbb{R}^{mxN} \quad (1.22)$$

$$J(e(w)) = \frac{1}{2} \cdot [Y^*(n) - Y(n)]^T \cdot [Y^*(n) - Y(n)] \quad (1.23)$$

$$Y(n) = w^T \cdot X(n) \quad (1.24)$$

$$J(e(w)) = \frac{1}{2} \cdot [Y^*(n) - w^T \cdot X(n)]^T \cdot [Y^*(n) - w^T \cdot X(n)] \quad (1.25)$$

$$J(e(w)) = \frac{1}{2} \cdot (Y^*(n))^T \cdot Y(n) - (Y^*(n))^T \cdot w^T \cdot X(n) + \frac{1}{2} \cdot (X(n))^T \cdot w \cdot w^T \cdot X(n) \quad (1.26)$$

Hallamos el valor óptimo de w es decir: $w^* = \text{argmin} J(e(w))$, calculamos la derivada de J respecto de w e igualamos a 0.

$$\begin{aligned} \frac{\partial}{\partial w} J(e(w)) &= \frac{\partial}{\partial w} \left(\frac{1}{2} \cdot (Y^*(n))^T \cdot Y(n) \right) - \frac{\partial}{\partial w} \left((Y^*(n))^T \cdot w^T \cdot X(n) \right) \\ &\quad + \frac{\partial}{\partial w} \left(\frac{1}{2} \cdot (X(n))^T \cdot w \cdot w^T \cdot X(n) \right) \end{aligned} \quad (1.27)$$

Eliminamos el primer término, ya que la derivada vale 0:

$$\frac{\partial}{\partial w} J(e(w)) = - \frac{\partial}{\partial w} \left((Y^*(n))^T \cdot w^T \cdot X(n) \right) + \frac{\partial}{\partial w} \left(\frac{1}{2} \cdot (X(n))^T \cdot w \cdot w^T \cdot X(n) \right) \quad (1.28)$$

Cambiamos el orden de las matrices:

$$\frac{\partial}{\partial w} J(e(w)) = - \frac{\partial}{\partial w} \left((X(n))^T \cdot w \cdot Y^*(n) \right) + \frac{\partial}{\partial w} \left(\frac{1}{2} \cdot (X(n))^T \cdot w \cdot w^T \cdot X(n) \right) \quad (1.29)$$

Aplicamos propiedad de la traspuesta del producto de matrices: $(A \cdot B)^T = B^T \cdot A^T$

$$\frac{\partial}{\partial w} J(e(w)) = \text{tr} \left\{ - \frac{\partial}{\partial w} \left(Y^*(n) \cdot (X(n))^T \cdot w \right) + \frac{\partial}{\partial w} \left(\frac{1}{2} \cdot w^T \cdot X(n) \cdot (X(n))^T \cdot w \right) \right\} \quad (1.30)$$

$$\frac{\partial}{\partial w} J(e(w)) = \text{tr} \left\{ - X(n) \cdot (Y^*(n))^T + X(n) \cdot (X(n))^T \cdot w \right\} \quad (1.31)$$

$$\frac{\partial}{\partial w} J(e(w)) = 0 \quad (1.32)$$

$$- X(n) \cdot (Y^*(n))^T + X(n) \cdot (X(n))^T \cdot w^* = 0 \quad (1.33)$$

$$X(n) \cdot (X(n))^T \cdot w^* = X(n) \cdot (Y^*(n))^T \quad (1.34)$$

$$w^* = \left[X(n) \cdot (X(n))^T \right]^{-1} \cdot X(n) \cdot (Y^*(n))^T \quad (1.35)$$

Dimensión de las matrices de la ecuación (1.35)

$$\mathbb{R}^m = [\mathbb{R}^{mxN} \mathbb{R}^{Nx m}] \quad [\mathbb{R}^{mxN} [\mathbb{R}^{1x N}]^T]$$

El proceso de aprendizaje de la red neuronal Adaline simple se describe con los siguientes pasos:

Paso 1: Inicializar los pesos y los sesgos de forma aleatoria

Paso 2: Tomar un patrón de entrenamiento con entrada $x(k)$

Paso 3: Calcular la salida $a = y(k)$, compararla con la salida deseada $d(k)$ y obtener la diferencia:

$$e(k) = d(k) - y(k) \quad (1.36)$$

Paso 4: Para todos los pesos y todos los sesgos, calcular:

$$w(k + 1) = w(k) + \gamma \cdot e(k) \cdot x(k) \quad (1.37)$$

$$b(k + 1) = b(k) + \gamma \cdot e(k) \quad (1.38)$$

Paso 6: Repetir los pasos anteriores: 2, 3, 4 y 5 para todos los datos de entrenamiento (1 vez)

Paso 7: Repetir los pasos 2, 3, 4, 5 y 6 la cantidad de ciclos hasta que se cumpla el criterio de detención

Los posibles criterios de detención para Adaline simple son los siguientes:

Criterio 1: Establecer un número máximo de ciclos. Este número debería garantizar que el error cuadrático de los patrones de entrenamiento se haya estabilizado.

Criterio 2: Cuando el error cuadrático en los patrones de entrenamiento no cambia durante cierta cantidad de ciclos

Criterio 3: Se puede utilizar un conjunto de datos de validación, que solo corresponde a una parte aleatoria del conjunto de entrenamiento. En este caso, el criterio sería cuando el error cuadrático en los patrones de validación no aumenta o permanece estable a lo largo de algunos ciclos (se establece una cantidad de ciclos adecuada).

Ejercicio 1.4

Red Neuronal Adaline

Analizar el siguiente ejemplo en Matlab®.

```
% Mediante la función linearlayer creamos una red ADALINE
net1 = linearlayer
% Mediante configure indicamos la estructura de la red para entrada [0;0] y salida 0
net1 = configure (net1, [1; 1], [1]);
% Esto indica que la red debe tener dos entradas y una salida.
% También se puede usar la función train, pero mediante configure podemos inspeccionar
% los pesos antes del entrenamiento.
% En forma predeterminada los sesgos y los pesos se inicializan en cero.
% Mostramos los pesos y el sesgo:
W = net1.IW {1,1}
% >>W =
%   0   0
b = net1.b {1}
% >> b =
%   0
% Se pueden asignar valores a los pesos (W) y al bias (b):
net1.IW {1,1} = [3 4];
net1.b {1} = -1;
% Mostramos la red
```

```

view(net1)
% Se puede simular ADALINE para un vector de entrada en particular.
p = [6; 6];
a = sim (net1, p)
% >>a =
% 41
% Graficamos
x=-8:0.01:8 ; y=1/4 -3/4*x ;
plot( x, y, 'linewidth',3); grid on

```

En la Fig. 22 mostramos la estructura de la red.

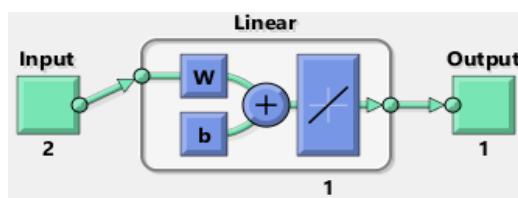


Fig. 22 – Estructura de la red ADALINE graficada con Matlab®

En resumen, se puede crear una red ADALINE con linearlayer, ajustar sus elementos como desee y simularlo con la función sim.

Teniendo en cuenta el ejemplo anterior, podemos graficar la recta de la siguiente manera

$$a = w_{11} \cdot p_1 + w_{12} \cdot p_2 + b \rightarrow w_{11} \cdot p_1 + w_{12} \cdot p_2 + b = 0$$

$$\text{Reemplazamos: } p_1 = x ; p_2 = y ; w_{11} = 3 ; w_{12} = 4 ; b = -1$$

$$\rightarrow 3 \cdot x + 4 \cdot y - 1 = 0 \rightarrow y = 1/4 - 3/4 \cdot x$$

Graficamos con Matlab®, en la Fig. 23 se muestran los resultados

$x=-8:0.01:8 ; y=1/4 -3/4*x ; \text{ plot}(x, y, \text{'linewidth'}, 3); \text{ grid on}$

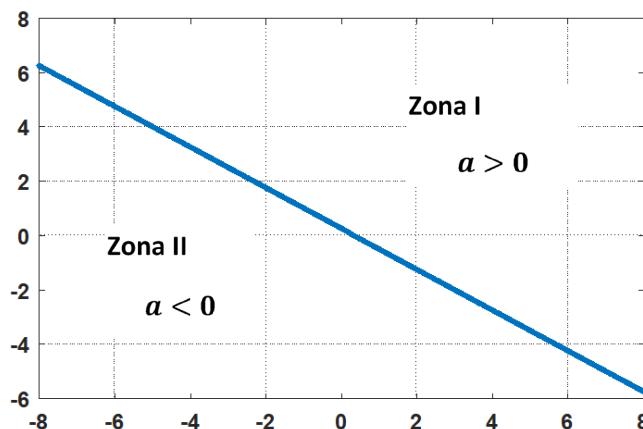


Fig. 23 – Ejemplo de gráfico simple con Matlab® para límite de decisión de una neurona simple ADALINE

La red neuronal Adaline puede resultar insuficiente para resolver problemas. Esto se debe a que tiene una sola capa (neurona) con función de salida lineal. Puede introducir funciones de salida que no sean lineales, pero LMS ya no es funcional. Además, tiene mala generalización. Por tal motivo se combinan redes Adaline en serie y en paralelo para armar la red Madaline (múltiple Adaline).

Las redes Madaline son unidades potentes de procesamiento, pero requieren más esfuerzos de diseño. Las funciones nuevas de activación (no lineales) complicaron los métodos de ajuste de pesos. Madaline son el punto de partida moderno para la aplicación de redes neuronales en inteligencia artificial.

Filtros adaptativos de redes neuronales

Las redes Adaline (neurona lineal adaptativa) son similares al perceptrón, pero se utiliza función de transferencia lineal en lugar función de limitador fuerte (hard-limiting). Esto permite que sus salidas tomen cualquier valor, mientras que la salida del perceptrón está limitada a 0 o 1. Tanto la red Adaline como la red neuronal perceptrón solo resuelven problemas que sean linealmente separables. Sin embargo, aquí se utiliza la regla de aprendizaje LMS (mínimos cuadrados medios), que es mucho más poderosa que la regla de aprendizaje utilizada en el perceptrón. La regla de aprendizaje LMS, o Widrow-Hoff, minimiza el error cuadrático medio y, por lo tanto, mueve los límites de decisión lo más lejos posible de los patrones de entrenamiento.

Un sistema lineal adaptativo con red neuronal responde a los cambios en su entorno mientras está en funcionamiento. Las redes lineales que se ajustan en cada paso de tiempo en función de vectores nuevos de entrada y de salida, pueden encontrar pesos y sesgos que minimicen la suma del error cuadrático medio de la red para los vectores de entrada y destino recientes. Estas redes se utilizan frecuentemente en sistemas de control, procesamiento de señales y cancelación de errores.

Los pioneros en este campo fueron Widrow y Hoff, quienes dieron el nombre de Adaline a los elementos lineales adaptativos.

Funciones adaptativas

La función adaptar, cambia los pesos y bias de una red de forma incremental durante el entrenamiento. La regla de Widrow-Hoff solo puede entrenar redes lineales de una sola capa. Sin embargo, esto no es una gran desventaja, ya que las redes lineales con solo una capa son igual de capaces que las redes lineales multicapas. Para cada red lineal de tipo multicapa, existe una red equivalente lineal de una sola capa.



Descarga de los códigos de los ejercicios

Referencias

- Beale, M. H., Hagan, M., & Demuth, H. (2020). Deep Learning Toolbox™ User's Guide. In MathWorks. <https://la.mathworks.com/help/deeplearning/index.html>
- Del Brío, M. B. y Molina S. (2007). Redes Neuronales y Sistemas Borrosos. 3ra edición. Ed. Alfaomega.
- Demuth H, Beale M, Hagan M. (2018). Neural Network Toolbox™ User's Guide Neural network toolbox. MathWorks.
- Hagan, M. T., Demuth, H. B., Beale, M. H., & De Jesus, O. (2014). Neural Network Design 2nd Edition. In Neural Networks in a Soft computing Framework.
- Haykin, S. (2008). Neural Networks and Learning Machines. In Pearson Prentice Hall New Jersey USA 936 pLinks (Vol. 3).
- Kaplunovich, A., & Yesha, Y. (2020). Refactoring of Neural Network Models for Hyperparameter Optimization in Serverless Cloud. Proceedings - 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW 2020. <https://doi.org/10.1145/3387940.3392268>
- Kundella, S., & Gobinath, R. (2020). Robust convolutional neural network for arrhythmia prediction in ECG signals. Materials Today: Proceedings. <https://doi.org/10.1016/j.matpr.2020.10.579>
- Pytorch, Biblioteca de aprendizaje automático Pytorch, 2021. <https://pytorch.org/>
- Tensorflow, Biblioteca de aprendizaje automático Tensorflow, desarrollada por Google, 2021, <https://www.tensorflow.org/>
- The Mathworks, Inc. MATLAB, Version 9.6, 2019. (2019). MATLAB 2019b - MathWorks. In www.Mathworks.Com/Products/Matlab.
- Mishra, P. (2019). PyTorch Recipes. In PyTorch Recipes. <https://doi.org/10.1007/978-1-4842-4258-2>
- Zed, A., S. (2015). Learn More Python 3 the Hard Way. CEUR Workshop Proceedings, 1542.

Capítulo II - Algoritmos de entrenamiento de redes: propagación hacia atrás y otros algoritmos

Se presentan diferentes algoritmos de entrenamiento de redes neuronales. Primero se explica el algoritmo de propagación hacia atrás, que es la base para entender el entrenamiento de las redes neuronales estáticas y dinámicas. Se utiliza la regla de la cadena para minimizar el error cuadrático medio. Se explica el problema de mínimos locales y sus posibles soluciones. Luego se presentan otros algoritmos, y por último se explica el algoritmo Levenberg Marquardt (LM) que es una combinación del método de Gauss-Newton y del descenso más pronunciado. Si bien este algoritmo tiene alta carga computacional, se obtienen muy buenos resultados. LM se utiliza por defecto en muchas plataformas, ya que presenta ventajas respecto de otros algoritmos. Por último, se muestran ejercicios analíticos y numéricos con predicciones, clasificaciones y evaluación de métricas para comparar modelos.

Introducción

El algoritmo de propagación hacia atrás, en inglés backpropagation (BP), es una generalización del algoritmo de mínimos cuadrados parciales: LMS (del inglés, Least-Mean-Square algorithm). Se utiliza para entrenar redes multicapa. Al igual que con la ley de aprendizaje de LMS, el algoritmo BP utiliza método de gradientes descendentes, en el que el índice de rendimiento es el error cuadrático medio (Beale, 2020).

El algoritmo LMS y el de propagación hacia atrás difieren en la forma que calculan las derivadas.

En las redes multicapa que utilizan funciones de transferencia no lineales, resulta complejo determinar la relación entre los pesos de la red y el error. Para calcular las derivadas, necesitamos usar la regla de la cadena.

Este algoritmo de propagación hacia atrás (Backpropagation) es muy utilizado, también es conocido como algoritmo de retropropagación, debido a que el error se propaga en dirección inversa al cálculo de las salidas. También se llama método del gradiente descendente, por la forma en que se obtiene el error. Este algoritmo minimiza el siguiente error cuadrático:

$$e^2 = [d(k) - y(k)]^2 \tag{2.1}$$

Una red neuronal prealimentada (conocida como feedforward) es una red donde las conexiones entre sus neuronas no se realimentan y no forman un ciclo. Resulta diferente a la red neuronal dinámica recurrente. La red más sencilla es la prealimentada.

En las redes neuronales prealimentadas con aprendizaje BP, es decir de propagación hacia atrás, se utilizan las estructuras con topologías Madaline, conformada por múltiples neuronas Adaline, con estructuras multicapas con respuestas paramétricas no lineales. Se utilizan funciones de activación lineales y no lineales. Estas redes son la inspiración para una amplia diversidad de redes neuronales modernas (Demuth, 2018).

Para aprendizajes no lineales, en la Fig. 24 se muestra la función de error cuadrático medio J en función de los pesos w para el caso de uno y 2 coeficientes w .

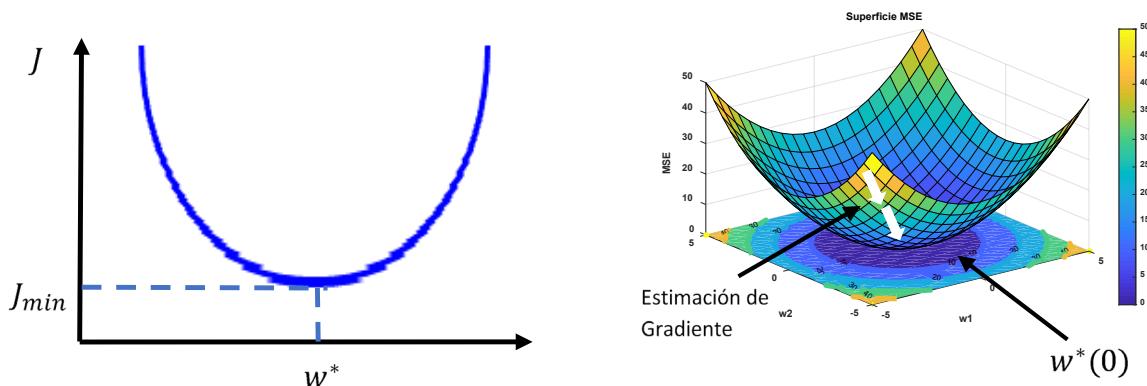


Fig. 24 – Función de error cuadrático medio J . Izquierda) en función de 1 coeficiente w , derecha) en función de 2 coeficientes w

Existen diferentes métodos de identificación paramétricos no lineales (The Mathworks, 2019)

- Gradiente descendente
- Método de los momentos de gradiente descendente
- Newton-Gauss
- Levenberg Marquardt
- Métodos de polinomios de alto orden

Método del Gradiente descendente para red neuronal de 2 capas

El método del gradiente descendente, también conocido como regla delta generalizada, calcula el gradiente negativo o gradiente descendente correspondiente al error cuadrático medio de la salida. Para luego ajustar los pesos de las neuronas. Primero se calcula el error en la capa de salida, con estos valores se calcula el error en la capa anterior, continuando así de adelante hacia atrás hasta llegar a la entrada; luego, se actualizan los pesos de cada capa de acuerdo con los valores de error calculados (Demuth, 2018). Para redes multicapa, no se puede utilizar el algoritmo Perceptron, debido a que no se conoce la respuesta correcta para las capas ocultas.

Teniendo en cuenta el error $e = d - y$, siendo:

y : salida de la neurona

d : salida deseada o correcta

Se define el error cuadrático medio con la siguiente ecuación:

$$E = \frac{1}{2} \cdot \sum_{..} (d^p - y^p)^2 \quad (2.2)$$

Se utiliza la minimización de errores mediante el método de gradiente descendente. Funciona correctamente si las funciones de transferencia de las neuronas son diferenciables. Los pesos de la red se actualizan con el siguiente incremento:

$$\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} \quad (2.3)$$

$$w(k+1) = w(k) + \Delta w_{ij} = w(k) - \eta \cdot \frac{\partial E}{\partial w_{ij}} \quad (2.4)$$

Primero analizamos una red neuronal de 1 capa, ver Fig. 25.

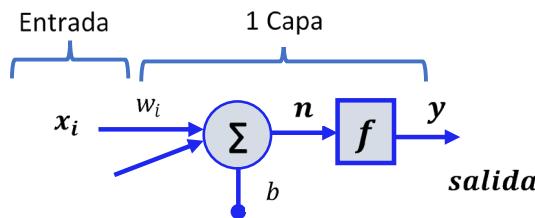


Fig. 25 – Red neuronal de una capa

Suponiendo función de transferencia lineal dada por $y = f(n) = n$, la salida y el error cuadrático medio están dados por las siguientes 2 ecuaciones:

$$y = n = \sum_i w_i \cdot x_i \quad (2.5)$$

$$E = \frac{1}{2} \cdot \sum_{..} (d^p - y^p)^2 \quad (2.6)$$

Derivando E en la ecuación anterior obtenemos:

$$\frac{dE}{dy} = y - d \quad (2.7)$$

$$\frac{\partial E}{\partial w_i} = \frac{dE}{dy} \cdot \frac{\partial y}{\partial w_i} = (y - d) \cdot x_i \quad (2.8)$$

$$\Delta w_i = -\eta \cdot \frac{\partial E}{\partial w_i} = -\eta \cdot (y - d) \cdot x_i \quad (2.9)$$

Extendemos el análisis para una red neuronal de 2 capas y funciones de transferencia f lineales y no lineales diferenciables.

$$n_j = \sum_i w_{ij} \cdot y_i \quad (2.10)$$

$$y_j = f(n_j) \quad (2.11)$$

En este ejemplo utilizamos $f(n) = \tanh(n)$, su derivada resulta $f'(n) = 1/\cosh^2(n)$

$$f'(n) = \frac{dy}{dn} = 1/\cosh^2 \left(\sum_i w_i \cdot x_i \right) \quad (2.12)$$

Derivando tenemos:

$$\frac{dy}{dn} = 1/\cosh^2(n) \quad (2.13)$$

$$\frac{dE}{dy} = y - d \quad ; \quad \frac{\partial n}{\partial w_i} = x_i \quad (2.14)$$

$$\frac{\partial E}{\partial w_i} = \frac{dE}{dy} \cdot \frac{dy}{dn} \cdot \frac{\partial n}{\partial w_i} \quad (2.15)$$

$$\frac{\partial E}{\partial w_i} = (y - d) \cdot \left[1/\cosh^2 \left(\sum_i w_i \cdot x_i \right) \right] \cdot x_i \quad (2.16)$$

Usamos la regla de la cadena para hallar la derivada del error en la red neuronal de la Fig. 26

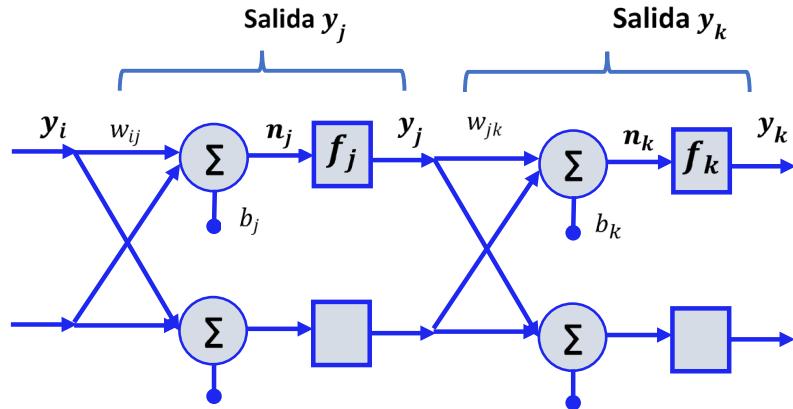


Fig. 26 – Red neuronal de 2 capas

En la capa con salida y_k , tenemos las siguientes ecuaciones:

$$\frac{dE}{dy_k} = y_k - d_k \quad (2.17)$$

$$\delta_k = \frac{\partial E}{\partial n_k} = \frac{dE}{dy_k} \cdot \frac{dy_k}{dn_k} = (y_k - d_k) \cdot f'(n_k) \quad (2.18)$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial n_k} \cdot \frac{\partial n_k}{\partial w_{jk}} = \frac{\partial E}{\partial n_k} \cdot y_j = \delta_k \cdot y_j \quad (2.19)$$

En la capa con salida y_j , tenemos las siguientes ecuaciones:

$$\frac{dE}{dy_j} = \sum_k \left(\frac{\partial E}{\partial n_k} \cdot \frac{\partial n_k}{\partial y_j} \right)$$

$$\delta_j = \frac{\partial E}{\partial n_i} = \frac{dE}{dy_i} \cdot f'(n_j) \quad (2.20)$$

$$\frac{\partial E}{\partial w_{ii}} = \frac{\partial E}{\partial n_i} \cdot \frac{\partial n_i}{\partial w_{ii}} = \frac{\partial E}{\partial n_i} \cdot y_i = \delta_j \cdot y_i \quad (2.21)$$

Para el ajuste de los pesos w utilizamos:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial n_k} \cdot \frac{\partial n_k}{\partial w_{jk}} = \delta_k \cdot y_j \quad ; \quad \frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial n_j} \cdot \frac{\partial n_j}{\partial w_{ij}} = \frac{\partial E}{\partial n_j} \cdot y_i = \delta_j \cdot y_i \quad (2.22)$$

$$\Delta w_{jk} = -\eta \cdot \frac{\partial E}{\partial w_{jk}} \quad (2.23)$$

$$\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} \quad (2.24)$$

$$w(k+1) = w(k) + \Delta w_{ij} \quad (2.25)$$

$$w(k+1) = w(k) - \eta \cdot \frac{\partial E}{\partial w_{ij}} \quad (2.26)$$

Los algoritmos pueden tener problemas con los mínimos locales, tales como se observa en la Fig. 27 donde se grafica el error cuadrático medio (MSE). Posiblemente el algoritmo encuentre un mínimo local y nunca encuentre el mínimo global o absoluto. En estos casos se suele agregar ruido para salir de esos mínimos locales y hallar el mínimo global. Otra opción es usar otros algoritmos para tratar de minimizar este problema, o usar redes neuronales con funciones de base radial (RBF).

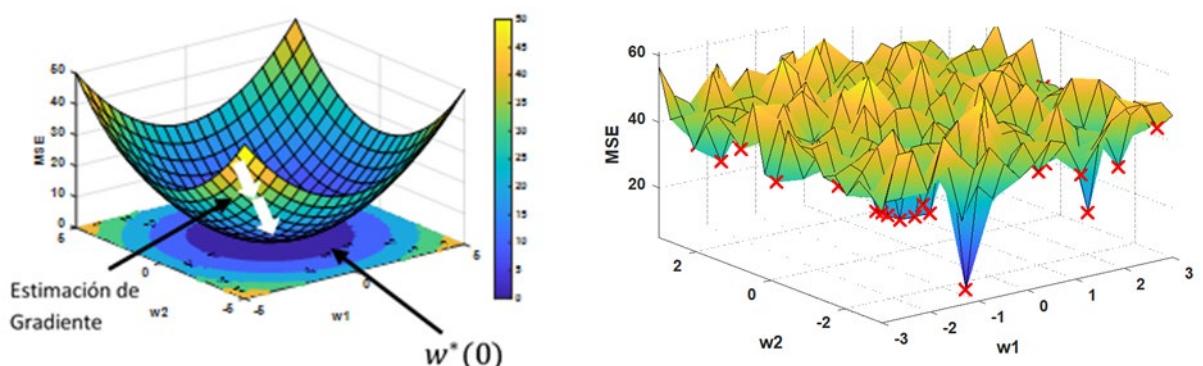


Fig. 27 – Superficie de error: función de error cuadrático medio J en función de w . Izquierda) Función convexa, derecha) problemas con mínimos locales

En la Fig. 28 se observa un ejemplo de trayectoria para 2 coeficientes, las circunferencias indican los valores de la función de error.

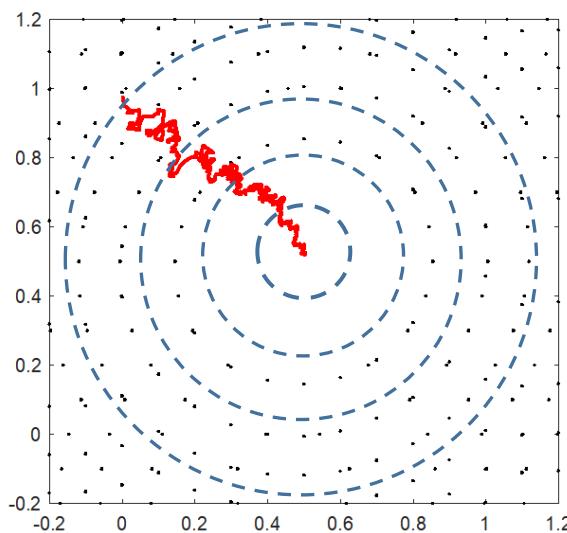


Fig. 28 – Ejemplo de trayectoria de 2 coeficientes W para minimizar el error cuadrático medio.

Método de gradiente descendente para múltiples capas

Por último, presentamos un resumen de las ecuaciones del método de gradiente descendente para múltiples capas en forma genérica (Hagan, 2014)

El algoritmo ajusta los parámetros para minimizar el error cuadrático medio $F(x)$

$$F(x) = E[e^T \cdot e] = E[(t - a)^T \cdot (t - a)] \quad (2.27)$$

La sensitividad en la última capa M resulta:

$$s_i^M = \frac{\partial F}{\partial n_i^M} = \frac{\partial(t - a)^T \cdot (t - a)}{\partial n_i^M} \quad (2.28)$$

Y la matriz de sensitividad resulta:

$$s^m = \frac{\partial F}{\partial n^m} \quad (2.29)$$

Siendo $g^1(n)$ la función de transferencia de la capa 1, y $g^2(n)$ la función de transferencia de la capa 2, sus derivadas en forma matricial resultan:

$$\mathbf{F}^1(\mathbf{n}^1) = \begin{bmatrix} f^1(n) & 0 \\ 0 & f^1(n) \end{bmatrix} = \begin{bmatrix} \frac{\partial g^1(n)}{\partial n} & 0 \\ 0 & \frac{\partial g^1(n)}{\partial n} \end{bmatrix} \quad (2.30)$$

$$\mathbf{F}^2(\mathbf{n}^2) = \begin{bmatrix} f^2(n) & 0 \\ 0 & f^2(n) \end{bmatrix} = \begin{bmatrix} \frac{\partial g^2(n)}{\partial n} & 0 \\ 0 & \frac{\partial g^2(n)}{\partial n} \end{bmatrix} \quad (2.31)$$

El primer paso es propagar las entradas hacia adelante a través de la red. Asignamos a^m a las salidas de las neuronas y p a las entradas, entonces a las entradas le podemos asignar a^0

$$a^0 = p \quad (2.32)$$

Siendo f^{m+1} la función de transferencia de la neurona de la capa $m + 1$, tenemos

$$a^{m+1} = f^{m+1}(W^{m+1} \cdot a^m + b^{m+1}) \quad (2.33)$$

para $m = 0, 1, \dots, M - 1$. La última salida corresponde $a = a^M$

El siguiente paso es propagar la sensitividad s hacia atrás a través de la red. Donde el vector t es la respuesta deseada de la red y n^M la salida la neurona de la capa M .

$$s^M = -2 \cdot \mathbf{F}^M(\mathbf{n}^M)(t - a) \quad (2.34)$$

$$s^m = \mathbf{F}^m(\mathbf{n}^m)(W^{m+1})^T s^{m+1} \quad (2.35)$$

para $m = M - 1, \dots, 2, 1$

Finalmente se ajusta los pesos y los sesgos mediante el gradiente descendente aproximado:

$$W^m(k + 1) = W^m(k) - \alpha \cdot s^m \cdot (a^{m-1})^T \quad (2.36)$$

$$b^m(k + 1) = b^m(k) - \alpha \cdot s^m \quad (2.37)$$

Mejoras en el rendimiento del algoritmo de propagación hacia atrás

Para analizar y mejorar el rendimiento del algoritmo de propagación hacia atrás (backpropagation) se deben tener en cuenta las siguientes consideraciones:

- Evitar los mínimos locales
- Evitar que las derivadas vayan a cero
- Aprovechar el impulso para acelerar el aprendizaje.
- Reducir la tasa de aprendizaje cuando los pesos oscilan.
- Utilizar pesos aleatorios iniciales pequeños y una tasa de aprendizaje inicial pequeña

Un problema con el algoritmo backpropagation son los mínimos locales, la superficie de error ya no tiene forma de taza.

El descenso en gradiente puede quedar atrapado en los mínimos locales. El "ruido" puede sacarnos de los mínimos locales, para esto se puede utilizar:

- Actualización estocástica (un patrón a la vez).
- Agregar ruido a los datos de entrenamiento, pesos o activaciones.
- Las tasas de aprendizaje elevadas pueden ser una fuente de ruido debido al sobre impulso.

Los objetivos de los clasificadores de 0 y 1 son inalcanzables por las funciones tanh. Los pesos aumentan a medida que el algoritmo intenta forzar a cada unidad de salida a alcanzar su valor asintótico. Al tratar de obtener un resultado correcto desde 0,95 hasta 1,0 se desperdicia tiempo y recursos que deberían concentrarse en otra parte. La solución es utilizar objetivos alcanzables de 0,1 y 0,9 en lugar de 0 y 1. Y no penalizar la red por sobrepasar estos objetivos.

Las señales de error δ se atenúan a medida que retroceden a través de múltiples capas. Entonces, diferentes capas aprenden a diferentes ritmos. Los pesos en las primeras capas ocultas demoran más tiempo en aprender que los pesos de las capas ocultas posteriores. Como solución hay que tener diferentes tasas de aprendizaje η para las diferentes capas.

Método del Gradiente descendente con velocidad de aprendizaje variable

Este algoritmo es una variación del método del gradiente descendente, para disminuir el número iteraciones sin alejarse de la solución. Se varía el factor de velocidad de aprendizaje η , según el error que se tenga en cada iteración. Si el error calculado en una iteración es menor que el error en la iteración anterior se aumenta en un factor determinado el valor de la velocidad de aprendizaje. En cambio, si el error calculado es igual o mayor al error en la iteración anterior la velocidad de aprendizaje disminuye. Este algoritmo tiene más pasos por iteración que la regla delta generalizada, ya que también debe comparar los valores de error para variar el factor de velocidad de aprendizaje.

Método del gradiente descendente con momentos

Este algoritmo también es una variación de método del gradiente descendente; este método agrega un término de momento a la actualización de pesos sinápticos, para acelerar la convergencia. El algoritmo es similar al algoritmo de gradiente descendente, pero la actualización de pesos es diferente (Hagan, 2018), (Mishra, 2019).

El aprendizaje resulta lento si la tasa de aprendizaje es muy baja. El gradiente puede ser pronunciado en algunas direcciones, pero poco profundo en otras. Como solución se agrega el término de momento a la actualización de los pesos sinápticos, acelerando la convergencia. La clave de esta optimización radica en actualizar los parámetros internos de la red sumando un término adicional que considera el valor de la última actualización de la iteración. Se tiene en cuenta los gradientes anteriores y también el gradiente actual. Se toma el gradiente actual y se multiplica por la tasa de aprendizaje η , y se suma el valor de la actualización anterior multiplicado por una constante conocida como el coeficiente de momento α . Esto se muestra en la siguiente ecuación:

$$\Delta w_{ij}(t) = -\eta \cdot \frac{\partial E}{\partial w_{ij}(t)} + \alpha \cdot \Delta w_{ij}(t-1) \quad (2.38)$$

Un valor típico de α puede ser 0,5. De esta forma, si la dirección del gradiente permanece constante, el algoritmo dará pasos cada vez más grandes. En este método aumenta el tiempo que dura cada iteración, pero converge a la solución en menor cantidad de iteraciones.

Método de Newton

Se conoce como algoritmo de segundo orden debido a que utiliza la matriz Hessiana. La matriz hessiana de una función o campo escalar es un matriz cuadrada de las segundas derivadas parciales. Este algoritmo busca las direcciones óptimas de variación de parámetros calculando las derivadas de orden 2 de la función de error. Utiliza el desarrollo de Taylor de segundo orden de f , se aproxima alrededor del conjunto inicial de pesos w_0 . Siendo H_0 el Hessiano del conjunto inicial w_0 (Beale, 2020).

$$f(w) = f_0 + g_0 \cdot (w - w_0) + \frac{1}{2} \cdot (w - w_0)^T \cdot H_0 \cdot (w - w_0) \quad (2.39)$$

Teniendo en cuenta que g es el gradiente de f y tiene que ser 0 para obtener el mínimo de f , tenemos la siguiente ecuación:

$$g(w) = g_0 + (w - w_0) \cdot H_0 \quad (2.40)$$

Se inicia con un vector de parámetros w_0 , mediante el método de Newton se obtiene la siguiente expresión para una red neuronal:

$$w_{i+1} = w_i - H_i^{-1} \cdot g_i \quad (2.41)$$

El método de Newton necesita menos pasos para converger respecto al método de gradiente descendente. Sin embargo, se requiere alta carga computacional para calcular la matriz Hessiana y su inversa.

Notas: El desarrollo de la serie de Taylor en un entorno del punto x_n se puede expresar mediante la siguiente ecuación:

$$f(x) = f(x_n) + f'(x_n) \cdot (x - x_n) + \frac{1}{2!} f''(x_n) \cdot (x - x_n)^2 + \dots \quad (2.42)$$

Si se trunca el desarrollo a partir del término de grado 2, y evaluamos en x_{n+1} . Y además x_{n+1} tiende a la raíz, es decir $f(x_n) = 0$, entonces de esta serie se obtiene el método de Newton que utiliza

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (2.43)$$

Donde se parte de la derivada

$$f'(x_n) = \frac{f(x_n)}{x_{n+1} - x_n} \quad (2.44)$$

Algoritmo de Levenberg Marquardt

El algoritmo de Levenberg-Marquardt, también se denomina método de mínimos cuadrados amortiguados. Consiste en una técnica iterativa que localiza el mínimo de una función, se utiliza en problemas de mínimos cuadrados con funciones no lineales (Lourakis, 2005). Está diseñado para acercarse en segundo orden, con entrenamientos rápidos sin necesidad de calcular la matriz Hessiana.

El algoritmo de Levenberg Marquardt funciona muy bien en las redes neuronales, generalmente presenta mejores resultados respecto de otros métodos.

Requiere gran esfuerzo computacional, pero es muy rápido y se utiliza por defecto en muchas plataformas para entrenar redes neuronales.

Resulta una combinación del método de Gauss-Newton y del descenso más pronunciado. Se diseñó para trabajar con funciones de error que se pueden expresar como suma de errores cuadráticos (Del Brío, 2007).

$$f(w) = \sum_{i=0}^m e_i^2(w) \quad (2.45)$$

Donde m es el largo de los datos de entrenamiento. No se necesita calcular la matriz Hessiana exacta, sino que se aproxima mediante la matriz Jacobiana J y el vector gradiente.

$$H \approx J^T \cdot J + \mu \cdot I \quad (2.46)$$

Este algoritmo se utiliza para encontrar las raíces de funciones formadas por la sumatoria de los cuadrados de funciones no lineales. Una aplicación especial de este algoritmo es el aprendizaje de redes neuronales. El algoritmo de Levenberg Marquardt se basa en el método iterativo de Newton y permite hallar las raíces de una función. Es decir, se utiliza para hallar los valores de las raíces; para el caso de las redes neuronales generalmente se utiliza la función de error cuadrático medio de las salidas de la red, donde las raíces de esta función son los valores correctos de los pesos W de las neuronas.

La actualización de la matriz de pesos (W) de las neuronas se calcula con la siguiente expresión:

$$W_{nuevo} = W_{actual} - [J^T \cdot J + \mu \cdot I]^{-1} \cdot J^T \cdot E \quad (2.47)$$

Siendo E es la matriz de errores en la red, I la matriz de identidad, μ es un valor escalar y J es la matriz Jacobiana, se calcula mediante la siguiente ecuación:

$$J = [\nabla E \cdot E^{-1}]^T \quad (2.48)$$

Siendo ∇E la matriz de gradientes de los errores.

Cada elemento de la matriz Jacobiana de la función de error se calcula mediante las derivadas de los errores respecto a los pesos, es decir:

$$J_{ij}(w) = \frac{\partial e_i}{\partial w_j} \quad (2.49)$$

Para redes neuronales donde se entrena con muchas entradas, conviene utilizar el método de Levenberg Marquardt ya que converge en menos iteraciones, a pesar de que cada iteración demore más tiempo.

Ejercicios Matlab®, Python y Analíticos

Ejercicio 2.1

Aproximación de funciones mediante el algoritmo de propagación hacia atrás

Mediante una red neuronal con una neurona en la capa de entrada, dos neuronas en la capa oculta y una en la capa de salida (1-2-1), ver Fig. 29, se quiere entrenar la misma para aproximar la siguiente función:

$$g(p) = 0,5 + \cos\left(\frac{\pi}{4} \cdot p\right) \text{ para entradas: } -3 \leq p \leq 3$$

Se entrena la red con muchos valores de p , mediante el esquema de la Fig. 30.

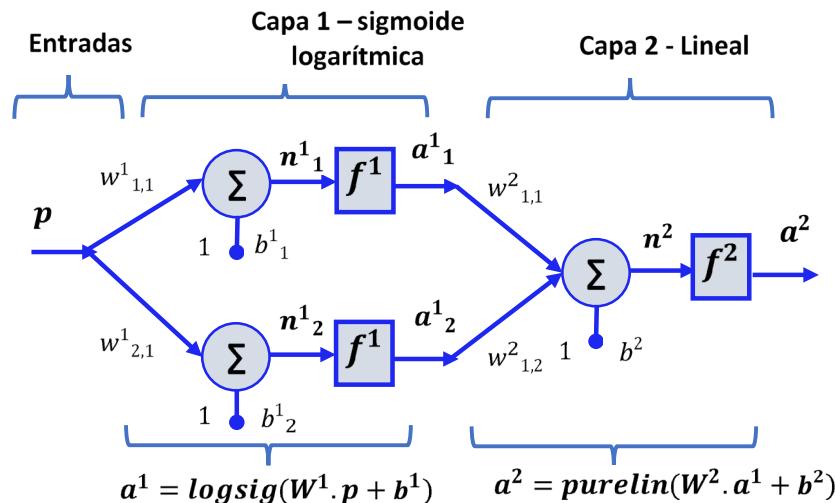


Fig. 29 – Ejemplo de aproximación de función mediante red neuronal 1-2-1

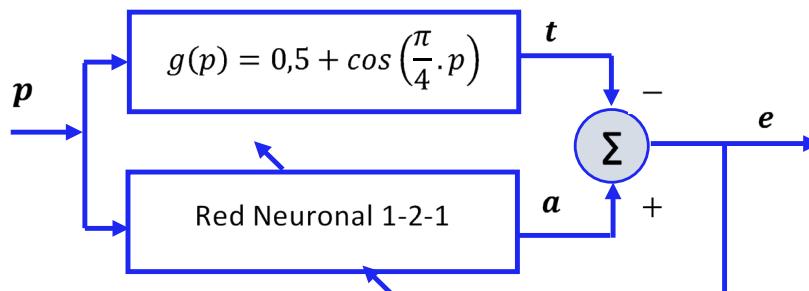


Fig. 30 – Esquema de adaptación mediante una red neuronal 1-2-1

Se pide:

Mediante el algoritmo de propagación hacia atrás (BP), ajustar los pesos en forma analítica para 1 solo punto de la entrada

Resolución

Se inicializan los valores de todos los pesos y de los sesgos, hay que asignar valores aleatorios bajos

$$W^1(0) = \begin{bmatrix} -0,15 \\ -0,1 \end{bmatrix}; \quad b^1(0) = \begin{bmatrix} -0,3 \\ -0,1 \end{bmatrix}; \quad W^2(0) = [0,11 \quad -0,15]; \quad b^2(0) = [0,4] \quad (2.50)$$

Se grafica en Matlab® la función que se quiere aproximar, ver Fig. 31.

```
p=-3:0.1:3; g = 0.5 + cos(pi/4*p); plot(p, g, 'linewidth',3); grid on
```

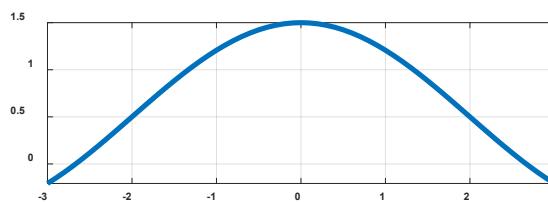


Fig. 31 – Función para aproximar con red neuronal

El conjunto de entrenamiento se puede definir mediante $\{p_1, t_1\}, \{p_2, t_2\}, \dots, \{p_n, t_n\}$. En este ejemplo, mediante el código Matlab® con intervalo de 0,1 se tienen 61 puntos de entrenamiento.

Para entrenar la red, en la primera iteración, se elige en forma aleatoria un punto de la entrada, por ejemplo $p = 1$. Con esta entrada calculamos la salida

$$a^0 = p = 1 \quad (2.51)$$

Luego calculamos el valor de la salida de la primera capa: a^1 y la salida: a^2 de la segunda capa.

$$a^1 = f^1(W^1 \cdot a^0 + b^1) = \text{logsig}\left(\begin{bmatrix} -0,15 \\ -0,1 \end{bmatrix} \cdot [1] + \begin{bmatrix} -0,3 \\ -0,1 \end{bmatrix}\right) \quad (2.52)$$

$$a^1 = \text{logsig}\left(\begin{bmatrix} -0,45 \\ -0,2 \end{bmatrix}\right) = \begin{bmatrix} \frac{1}{1 + e^{-0,45}} \\ \frac{1}{1 + e^{-0,2}} \end{bmatrix} \quad (2.53)$$

$$a^1 = \begin{bmatrix} 0,389 \\ 0,45 \end{bmatrix} \quad (2.54)$$

$$a^2 = f^2(W^2 \cdot a^1 + b^2) = \text{purelin}\left([0,11 \ - 0,15] \cdot \begin{bmatrix} 0,389 \\ 0,4502 \end{bmatrix} + [0,4]\right) = [0,3753] \quad (2.55)$$

Se calcula el error

$$e = t - a^{\text{capa } 2} = \left\{ 0,5 + \cos\left(\frac{\pi}{4} \cdot p\right) \right\} - a^2 \quad (2.56)$$

$$e = \left\{ 0,5 + \cos\left(\frac{\pi}{4} \cdot 1\right) \right\} - 0,3753 = 0,8318 \quad (2.57)$$

Ahora hay que propagar hacia atrás la sensitividad. Mediante estas ecuaciones:

$$s^M = -2 \cdot \mathbf{F}^M(\mathbf{n}^M)(t - a) \quad (2.58)$$

$$s^m = \mathbf{F}^m(\mathbf{n}^m)(W^{m+1})^T s^{m+1} \quad (2.59)$$

Debemos calcular las derivadas de las funciones de transferencia de las neuronas, se utilizan función sigmoide logarítmica y lineal

$$f^1(n) = \frac{d}{dn} \left(\frac{1}{1 + e^{-n}} \right) = \frac{e^{-n}}{(1 + e^{-n})^2} = \left(1 - \frac{1}{1 + e^{-n}} \right) \cdot \frac{1}{1 + e^{-n}} = (1 - a^1) \cdot a^1 \quad (2.60)$$

$$f^2(n) = \frac{d}{dn}(n) = 1 \quad (2.61)$$

El punto de inicio para propagar hacia atrás es el error $e = t - a = 0,8318$. Calculamos la sensitividad de la capa 2

$$s^2 = -2 \cdot \mathbf{F}^2(\mathbf{n}^2)(t - a) = -2 \cdot [f^2(n^2)] \cdot (0,8318) = -2 \cdot [1] \cdot (0,8318) = -1,6636 \quad (2.62)$$

Calculamos la sensitividad de la capa 1

$$s^1 = \mathbf{F}^1(\mathbf{n}^1) \cdot (W^2)^T \cdot s^2 = \begin{bmatrix} (1 - a_1^1) \cdot a_1^1 & 0 \\ 0 & (1 - a_2^1) \cdot a_2^1 \end{bmatrix} \cdot \begin{bmatrix} 0,11 \\ -0,15 \end{bmatrix} \cdot [-1,6636] \quad (2.63)$$

$$s^1 = \begin{bmatrix} (1 - 0,3894) \cdot 0,3894 & 0 \\ 0 & (1 - 0,45) \cdot 0,45 \end{bmatrix} \cdot \begin{bmatrix} 0,11 \\ -0,15 \end{bmatrix} \cdot [-1,6636] \quad (2.64)$$

$$s^1 = \begin{bmatrix} 0,2378 & 0 \\ 0 & 0,2475 \end{bmatrix} \cdot \begin{bmatrix} -0,183 \\ 0,2495 \end{bmatrix} \quad (2.65)$$

$$s^1 = \begin{bmatrix} -0,0435 \\ 0,0618 \end{bmatrix} \quad (2.66)$$

Por último hay que ajustar los pesos y los sesgos de las neuronas mediante las siguientes ecuaciones

$$W^m(k+1) = W^m(k) - \alpha \cdot s^m \cdot (a^{m-1})^T \quad (2.67)$$

$$b^m(k+1) = b^m(k) - \alpha \cdot s^m \quad (2.68)$$

Adoptamos $\alpha = 0,1$ y reemplazamos:

$$W^2(1) = W^2(0) - \alpha \cdot s^2 \cdot (a^1)^T = [0,11 \ -0,15] - 0,1 \cdot [-1,6636] \cdot [0,3894 \ 0,45] \quad (2.69)$$

$$W^2(1) = [0,11 \ -0,15] + [0,16636] \cdot [0,3894 \ 0,45] \quad (2.70)$$

$$W^2(1) = [0,1748 \ -0,0751] \quad (2.71)$$

$$b^2(1) = b^2(0) - \alpha \cdot s^2 = [0,4] - 0,1 \cdot [-1,6636] = [0,5664] \quad (2.72)$$

$$W^1(1) = W^1(0) - \alpha \cdot s^1 \cdot (a^0)^T = \begin{bmatrix} -0,15 \\ -0,1 \end{bmatrix} - 0,1 \cdot \begin{bmatrix} -0,0435 \\ 0,0618 \end{bmatrix} \cdot [1] = \begin{bmatrix} -0,1457 \\ -0,1062 \end{bmatrix} \quad (2.73)$$

$$b^1(1) = b^1(0) - \alpha \cdot s^1 = \begin{bmatrix} -0,3 \\ -0,1 \end{bmatrix} - 0,1 \cdot \begin{bmatrix} -0,0435 \\ 0,0618 \end{bmatrix} = \begin{bmatrix} -0,2956 \\ -0,1062 \end{bmatrix} \quad (2.74)$$

Con este paso se completa la primer iteración. Posteriormente hay que tomar otro punto de la entrada en forma aleatoria y volver a ajustar los pesos. Se debe continuar hasta tener un error aceptable, o hasta que se haya cumplido una cantidad máxima de iteraciones.

Ejercicio 2.2

Aproximación de funciones con Matlab mediante algoritmo de propagación hacia atrás. Funciones con ruido

Se pide aproximar la función cosenoidal $g(p)$ mediante una red neuronal 1-3-1, ver Fig. 32.

$$g(p) = 0,6 + \cos\left(\frac{\pi}{6} \cdot p\right) + \text{ruido} \quad \text{para } 0 \leq p \leq 10$$

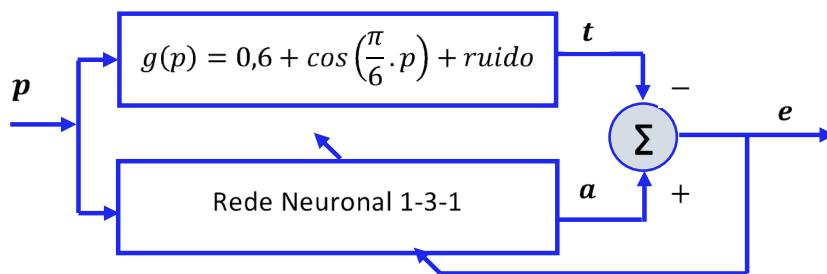


Fig. 32 – Diagrama de adaptación de la red neuronal 1-3-1

Resolución

```

%% Armamos el conjunto de datos
x= 0: 0.02:10 ;
t_set = 0.6 +cos((pi/6)*x);
t= t_set;
t= t + 0.15 *randn(1,length(t)) ; % Sumamos ruido a la salida.
%% Construimos la red neuronal de alimentación hacia adelante (feedforward)
red1 = feedforwardnet(3);
%% Seleccionamos las funciones que queremos graficar
% Tipear en línea de comandos: help nnplot
red1.plotFcns = {'plotperform', 'ploterrhist', 'plotfit', 'plotregression', 'plottrainstate'} ;
%% Mediante train, entrenamos la red neuronal
[red1,tr] = train(red1,x,t);
view(red1) % Mostramos la red
%% Testeamos la red
salida1 = red1(x);
e1 = gsubtract(t,salida1);
performance = perform(red1, t, salida1)
%% Graficamos los resultados
figure; plotperform(tr) ; figure; plottrainstate(tr);
figure; ploterrhist(e1) ; figure; plotregression(t,salida1);
figure; plotfit(red1, x, t);

```

En la Fig. 33 mostramos los resultados obtenidos, se observa un coeficiente de correlación alto: $R = 0.97394$

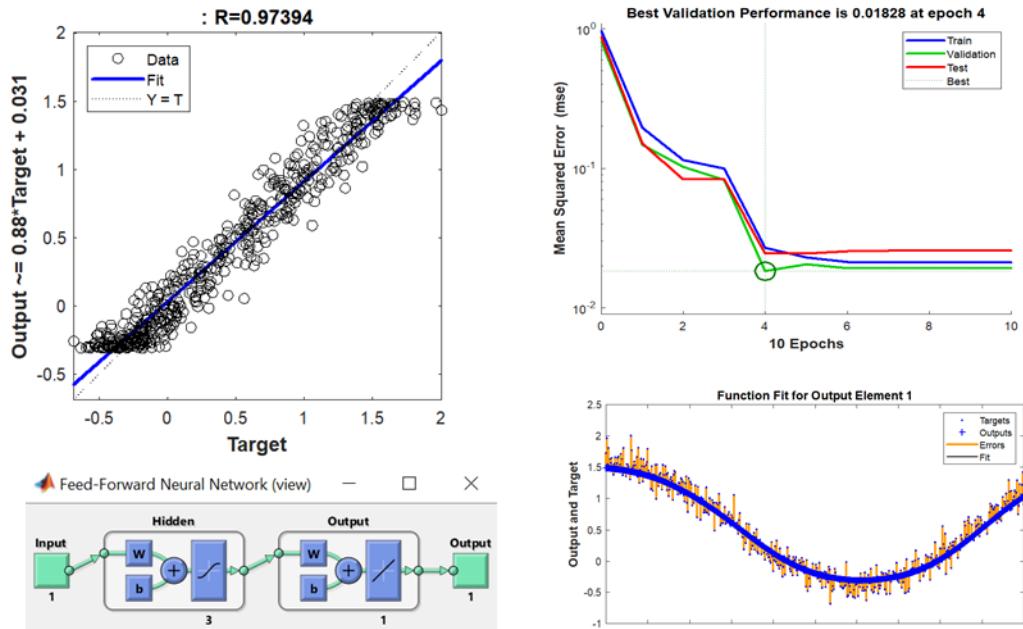


Fig. 33 – Resultados obtenidos de aproximación de función coseno con ruido mediante red neuronal

Ejercicio 2.3

Aproximación de funciones con algoritmo de propagación hacia atrás

Analizar el funcionamiento del siguiente código. Modificar para 5 neuronas y comparar resultados

```
% Ejemplo de red neuronal feedforward para resolver un problema simple.
% Cargamos los datos de entrenamiento
[ x, t ] = simplefit_dataset;
% Se tiene una matriz x de 1-x-94 con los valores de entrada y otra matriz t
% de 1-x-94 que contiene los valores de las respuestas asociadas
% Construimos una red del tipo feedforward, con una capa oculta de 8 neuronas.
net1 = feedforwardnet(8); % En forma predeterminada utiliza el algoritmo Levenberg-Marquardt
% Entrenamos la red con los datos de entrenamiento
net1 = train(net1,x,t); % En forma predeterminada utiliza el algoritmo Levenberg-Marquardt
view(net1) % Mostramos la red
% Estimamos las respuestas mediante la red entrenada
y = net1(x);
% Evaluamos el desempeño de la red. La función de rendimiento predeterminada es el error
% cuadrático medio.
perf = perform(net1,y,t)
% Repetimos con 5 neuronas
```

```
clear all ; close all
[x,t] = simplefit_dataset;
net2 = feedforwardnet(5);
net2 = train(net2,x,t);
view(net2) % Mostramos la red
y = net2(x) ;
perf = perform(net2, y, t)
```

Mediante la herramienta nntraintool de Matlab® se puede observar la performance obtenida de la red neuronal, basada en el error cuadrático medio

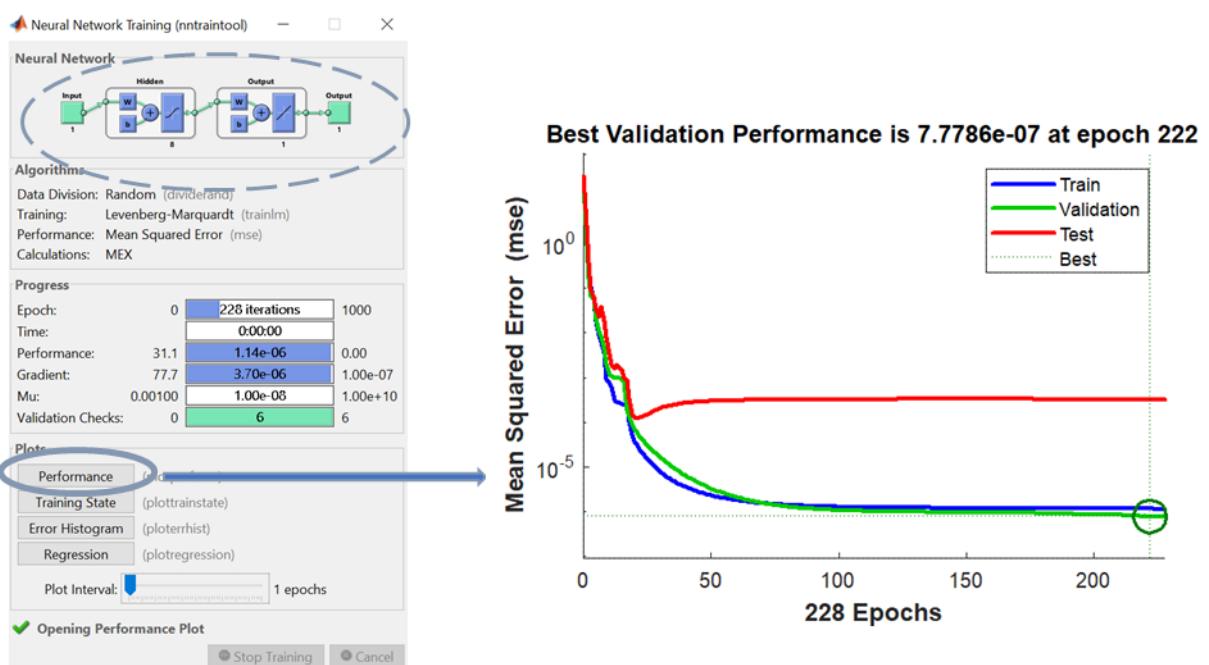


Fig. 34 – Resultados de red neuronal de propagación hacia atrás con 8 neuronas en la capa oculta.

Ejercicio 2.4

Ajuste de datos mediante una Red Neuronal estática.

Mediante Matlab® se pide construir una red neuronal del tipo alimentación hacia adelante (feedforward) para ajustar conjunto de datos senoidales con ruido

```
%% Red neuronal simple para ajustar datos
% Podemos importar los datos: x entradas y t salidas
% load mis_datos1; x = inputs ; t = targets ;
%% Armamos el conjunto de datos
% vector x: entradas, vector t: salidas deseadas
x=0;
for i=1:99
```

```

dx = abs(0.01*randn()) ;
x=[x (x(end)+ dx)] ;
end
t= sin(20*x) +cos(8*x) +1; % suma de seno y coseno
t= t + 0.2 *randn(1,length(t)) ; % Sumamos ruido
figure; plot(x,t, 'linewidth',2); axis tight; grid on
%% Con fitnet creamos la red neuronal para ajustar datos (regresión)
% para ayudas tipear help nntrain
% 'trainlm' corresponde al algoritmo Levenberg-Marquardt de propagación hacia atrás,
% generalmente rápido.
% 'trainbr' utiliza algoritmo de regulación Bayesiana. Lento, pero con mejores resultados
% en problemas difíciles.
% 'trainscg' basado en algoritmo de propagación hacia atrás con gradiente conjugado.
% 'trainscg' utiliza poca memoria.
trainFcn = 'trainlm'; % elegimos algoritmo Levenberg-Marquardt backpropagation.
hiddenLayerSize = 8;
net2 = fitnet(hiddenLayerSize,trainFcn);
%% Asignamos las funciones para pre y post procesamiento de entradas y salidas
% Se puede mostrar la lista de todas las funciones con los comandos: help nnprocess
net2.input.processFcns = { 'removeconstantrows' , 'mapminmax' }; %
net2.output.processFcns = { 'removeconstantrows' , 'mapminmax' }; %
%% Porcentajes de datos de entrenamiento, de validación y de testeo
% Tipear para ver ayudas: help nndivision
net2.divideFcn = 'dividerand'; % Divide los datos en forma aleatoria
net2.divideMode = 'sample'; % Divide cada muestra
net2.divideParam.trainRatio = 80/100;
net2.divideParam.valRatio = 10/100;
net2.divideParam.testRatio = 10/100;
%% Seleccionamos 'mse' función de Performance. Para ayudas tipear: help nnperformance
net2.performFcn = 'mse'; % Error cuadrático medio
%% Seleccionamos las funciones a graficar
% Para ayudas tipear: help nnplot
net2.plotFcns = { 'plotperform', 'plotregression','plottrainstate', 'plotfit', 'ploterrhist' };
%% Entrenamos la red
[net2,tr] = train(net2,x,t);
%% Testeamos la red
y1 = net2(x);
e1 = gsubtract(t,y1);
Performance1 = perform(net2,t,y1)
%% Recalculamos Performance de Entrenamiento, Validación y testeo
train_TTargets1 = t .* tr.trainMask{1};
val_TTargets1 = t .* tr.valMask{1};
test_TTargets1 = t .* tr.testMask{1};
% Mostramos performance
train_Performance1 = perform(net2,train_TTargets1,y1)
val_Performance1 = perform(net2,val_TTargets1,y1)
test_Performance1 = perform(net2,test_TTargets1,y1)
%% Mostramos la red neuronal
view(net2)
%% Graficamos

```

```
% Agregar comentarios dependiendo de los resultados que se quieran mostrar
figure; plotperform(tr) ; figure; plottrainstate(tr) ;
figure; ploterrhist(e1) ; figure; plotregression(t,y1) ;
figure; plotfit(net2,x,t);

%% Desarrollo, generamos script (en forma opcional)
% Generamos script con función de red neuronal
genFunction(net2,'NeuralNetworkFunction1') ;
y = NeuralNetworkFunction1(x);

% Generamos otra función de red neuronal, solo soporta matrices de entrada (no array de celdas)
genFunction(net2,'NeuralNetworkFunction1', 'MatrixOnly', 'yes') ;
y = NeuralNetworkFunction1(x);

% Generamos diagrama Simulink
% gensim(net2);
```

En la Fig. 52 se muestran los resultados del ajuste de datos. También se observa la topología que tiene la red. Con los datos utilizados, se obtuvo un alto coeficiente de correlación para todos los datos: $R = 0,98621$

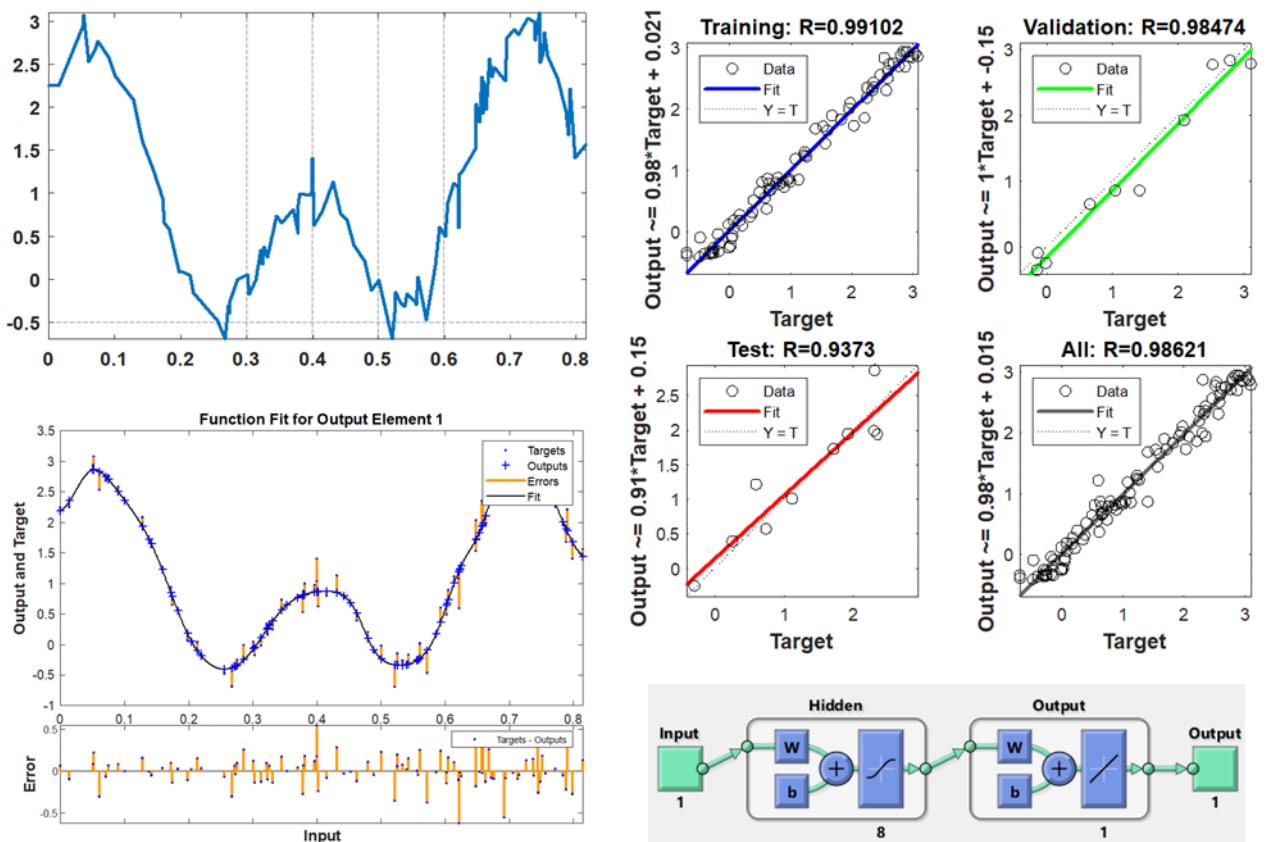


Fig. 35 – Resultados del Ajuste de datos con redes neuronales estáticas

Ejercicio 2.5

Clasificación de datos mediante red neuronal MLP (Perceptron Multi Capa). Matriz de confusión y métricas. En lenguaje Python.

Se utilizan librerías de scikit-learn en Python (Scikit-learn, 2021).

Se pide:

- Probar y analizar el funcionamiento del programa.
- Modificar parámetros y analizar los cambios en el rendimiento de la red.

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt # Librería gráfica
import numpy as np
# Definimos marcadores y colores de gráficos
colores= ['g', 'r', 'b', 'y', 'y']
marcadores = [ 'o' , (5,1), ',', '^','v', '<', '>', 's', 'd', '.', 'x', '+']
# Armamos el conjunto de datos y lo graficamos
nro_muestras = 320
centros_1 = ( [1, 4.2], [-1.6, 1], [-1.4, 3.7],[1.5, 1.5],[0, 7.5])
datos_1, labels = make_blobs(n_samples=nro_muestras,
    centers=centros_1, cluster_std=0.7, #0.65
    random_state = 0 )
colores = ('red', 'blue', 'green', 'orange', 'magenta')
plt.rc('axes', titlesize=18) ; plt.rc('font', size=16) ;
fig, ax1 = plt.subplots( figsize = (8,4) )
for n_clases1 in range(len(centros_1)):
    ax1.scatter(datos_1[labels==n_clases1][:, 0],
        datos_1[labels==n_clases1][:, 1],
        c = colores[n_clases1], s=60, label=str(n_clases1) ,
        marker = marcadores[n_clases1] )
ax1.grid() ; ax1.legend() ;
# Separamos los datos en testeо y entrenamiento
from sklearn.model_selection import train_test_split
dataset1 = train_test_split(datos_1, labels, test_size=0.15)
train_data1, test_data1, train_labels1, test_labels1 = dataset1
#####
### Construimos la red MLP para clasificar
from sklearn.neural_network import MLPClassifier
alpha1 = 1e-5
clf1 = MLPClassifier(solver='lbfgs', # Método de Newton
    alpha = 1e-5, # Parámetro L2 de penalización
    hidden_layer_sizes = (5,), # cantidad de capas ocultas, probar otros valores
    random_state =0 ) # Constante para generación de valores aleatorios de w y b
# Con el siguiente código simple se puede generar red, pero se generan redes grandes
# clf1 = MLPClassifier( random_state=1, max_iter=500 ).fit(train_data1, train_labels1)
# clf1.get_params( 1 )
# Entrenamos la red MLP
clf1.fit(train_data1, train_labels1)
```

```

#####
### Testeamos la red MLP
# Hallamos el valor medio de precisión
prec =clf1.score(train_data1, train_labels1)
print( "Valor medio de Precisión para los datos de prueba: ", prec )
# Clasificamos todos los datos y calculamos precisión de testeo y de entrenamiento
from sklearn.metrics import accuracy_score
predict_test1 = clf1.predict(test_data1)
predict_train1 = clf1.predict(train_data1)
train_precision1 = accuracy_score(predict_train1, train_labels1)
print( " Precisión para los datos de entrenamiento: ", train_precision1)
test_precision1 = accuracy_score(predict_test1, test_labels1)
print("Precisión para datos de prueba: ", test_precision1)
# Mostramos solo lo primeros 10 resultados predecidos
a = predict_train1[:10]
print( a )
# Hallamos Matriz de confusión
from sklearn.metrics import confusion_matrix
conf_matr_train1 = confusion_matrix(predict_train1, train_labels1)
conf_matr_test1 = confusion_matrix(predict_test1, test_labels1)
print(conf_matr_train1)
print(conf_matr_test1)
from sklearn.metrics import classification_report
print( classification_report(predict_test1, test_labels1) )
print( classification_report(predict_train1, train_labels1) )

#####
### Graficamos en 2D con diferentes regiones para distintos parámetros
vector_alpha1 =[1e-5, 1e-2 ,0.1 ,1]
# Generamos grilla para graficar
x_min1, x_max1 = datos_1[:, 0].min() - 0.45 , datos_1[:, 0].max() + .45
y_min1, y_max1 = datos_1[:, 1].min() - 0.45 , datos_1[:, 1].max() + .45
h = .02 # paso de la grilla
xx, yy = np.meshgrid( np.arange(x_min1, x_max1, h ),
                      np.arange(y_min1, y_max1, h ) )
for alpha1 in vector_alpha1:
    clf2 = MLPClassifier(solver='lbfgs', # Método de Newton
                          alpha=alpha1, # Parámetro L2 de penalización.
                          hidden_layer_sizes=(5,), # cant. de capas ocultas, probar otros valores
                          max_iter = 2000,
                          random_state =0 ,) # Generación de valores aleatorios para w y b
    # Entrenamos la red MLP
    clf2.fit(train_data1, train_labels1)
    # Clasificamos toda la grilla y graficamos distintas zonas
    Z = clf2.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    fig, ax3 = plt.subplots( figsize=(8,4) )
    niveles = [0, 1, 2, 3, 4]
    cs1 = ax3.contourf(xx, yy, Z, niveles,

```

```

colors=( 'g', 'r', 'b', 'y', 'c'),
origin='lower', alpha=.5 , extend='both' )
plt.colorbar(cs1)
# Graficamos los datos
colores= [ 'g', 'r', 'b', 'y', 'y']
marcadores = [ 'o', (5,1), '.', 'v', '^', '<', 's', '>', 'd', '!', 'x', '+']
for n_clases1 in range(len(centros_1)):
    ax3.scatter(datos_1[ labels==n_clases1 ][:, 0],
                datos_1[labels==n_clases1][:, 1],
                c=colores[n_clases1], s=60, label=str(n_clases1) ,
                marker= marcadores[n_clases1] ) #
ax3.set_xlim(xx.min(), xx.max())
ax3.set_ylim(yy.min(), yy.max())
ax3.set_title('Clasificación') ;ax3.grid()
ax3.set_xticks() , ax3.set_yticks()

```

Resultados

Valor medio del valor de Precisión para los datos de prueba: 0.9632

Valor medio de Precisión para datos de entrenamiento: 0.9632

Precisión para datos de prueba: 1.0

Mostramos solo los 10 primeros resultados de la clasificación

[4 3 4 1 4 0 1 4 0 0]

En la Tabla III mostramos matriz de confusión para los datos de entrenamiento. En la Tabla IV mostramos matriz de confusión para los datos de prueba

Tabla III – Matriz de Confusión del entrenamiento

Clase (grupo)	0	1	2	3	4
0	53	0	1	1	2
1	0	53	1	1	0
2	1	0	50	0	0
3	1	1	0	54	0
4	1	0	0	0	52

Tabla IV – Matriz de Confusión para los datos de prueba

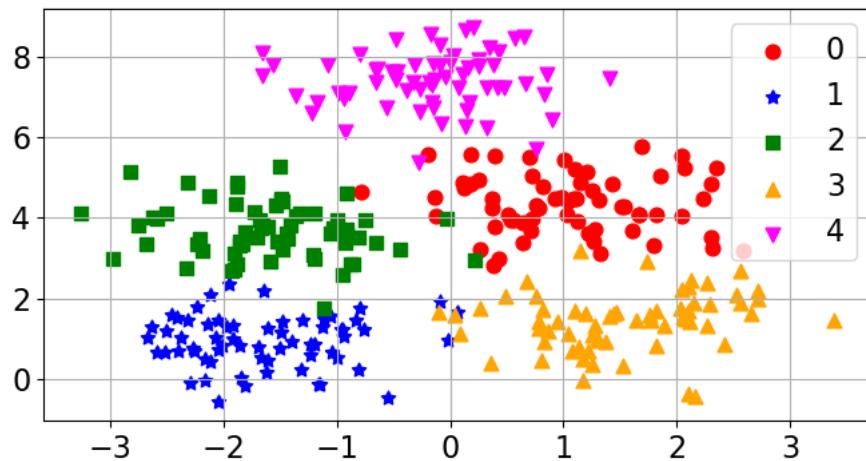
Clase	0	1	2	3	4
0	8	0	0	0	0
1	0	10	0	0	0
2	0	0	12	0	0
3	0	0	0	0	10
4	0	0	0	0	7

En la Tabla V se muestra el reporte de clasificación correspondiente a los datos de prueba

Tabla V – Reporte de Clasificación

Clase	Precisión	Sensibilidad	Medida F1	soporte
0	0.95	0.93	0.94	57
1	0.98	0.96	0.97	55
2	0.96	0.98	0.96	51
3	0.96	0.96	0.96	56
4	0.96	0.98	0.97	53

En la Fig. 36 se muestran todos los datos. En las Fig. 37 y Fig. 38 se muestran los resultados de la clasificación para distintos valores del parámetro alpha utilizado en la red neuronal.

*Fig. 36 –Gráfico de datos de entrada con 5 categorías*

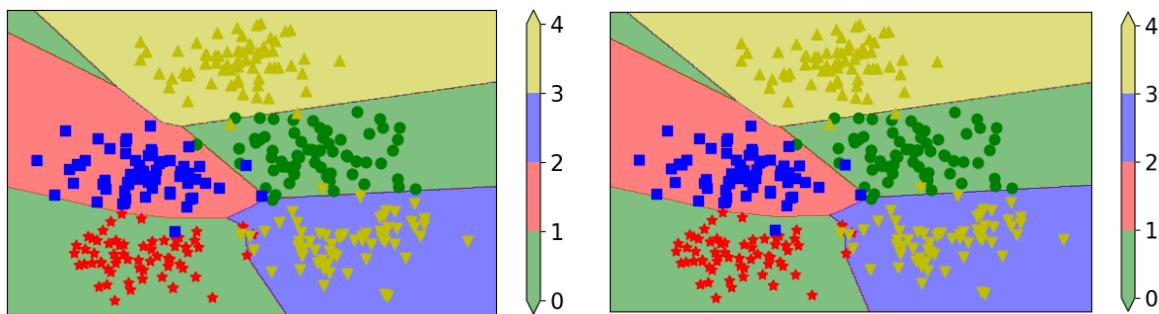


Fig. 37 – Clasificación de datos. Izq.) coeficiente alpha= 1e-5, derecha) alpha=1e-2

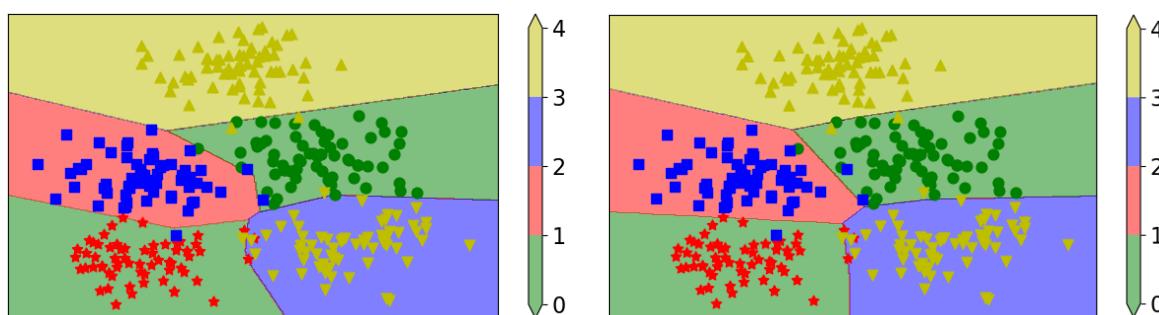


Fig. 38 – Clasificación de datos. Izq.) coeficiente alpha= 0,1, derecha) alpha=1

Ejercicio 2.6

Ajuste de datos con las librerías scikit-learn de Python

Se muestra un ejemplo de ajuste de datos aleatorios (Pytorch, 2021). Se pide analizar el funcionamiento del programa y luego modificar parámetros para mejorar los resultados.

```
from sklearn.model_selection import train_test_split # Importamos librerías
from sklearn.neural_network import MLPRegressor # librerías MLP
from sklearn.datasets import make_regression # librerías Regresión
import numpy as np # librerías numpy
import matplotlib.pyplot as plt
# Podemos usar esta función para generar los datos
# X, y = make_regression( n_samples= 320, random_state=1 )
# Generamos los datos de muestra
largo1 = 320
np.random.seed(seed=1)
mu1, sigma1 = 0, 1 # mu1: valor medio y sigma1: desviación estándar
# x1: Matriz de datos, tamaño: largo1 x 3.
# Generamos ruido aleatorio con distribución normal (gaussiana)
X = np.random.normal( mu1, sigma1, (largo1, 2) )
# Ruido v1 para agregar
v1 = np.random.normal( 0.5, 0.5, largo1 )
```

```

# Señal original X y Señal deseada: y
y = 3*X[:,0] - 1.9*X[:,1] #+ 0.9*X[:,2]
y = y + 2.9*v1 # agregamos ruido
# Separamos en entrenamiento y prueba
X_train1, X_test1, y_train1, y_test1 = train_test_split(X, y, random_state=1)
regr1 = MLPRegressor(random_state=1, max_iter=800).fit(X_train1, y_train1)
resu1 = regr1.predict(X_test1[:,2])
resu2 = regr1.score(X_test1, y_test1)
print(resu1) ; print(resu2)
# Predecimos datos
y_test1_predecido = regr1.predict(X_test1)
# Graficamos
plt.rc('font', size=16) ; plt.rc('axes', titlesize=18)
fig, ax1 = plt.subplots( figsize = (8 ,4) ) #
n = np.arange( (len(y_test1)) )
marcadores = ['o', (5,1), ',', '^', 'v', '<', '>', 'd', '.', 's', 'x', '+']
ax1.plot(n, y_test1 ,label= 'Valor Real')
ax1.plot(n, y_test1_predecido ,label= 'Valor Predecido')
ax1.grid() ;ax1.legend()
ax1.set_xlabel('Número de muestra');
plt.tight_layout()

```

Resultados

Se muestran los valores predecidos para 2 muestras:

[4.50867823 5.12465358]

Puntuación de confianza obtenida para los datos de prueba

0.8316

En la Fig. 39 se muestran los valores predecidos y los valores reales de las muestras, se observa un correcto ajuste de los datos.

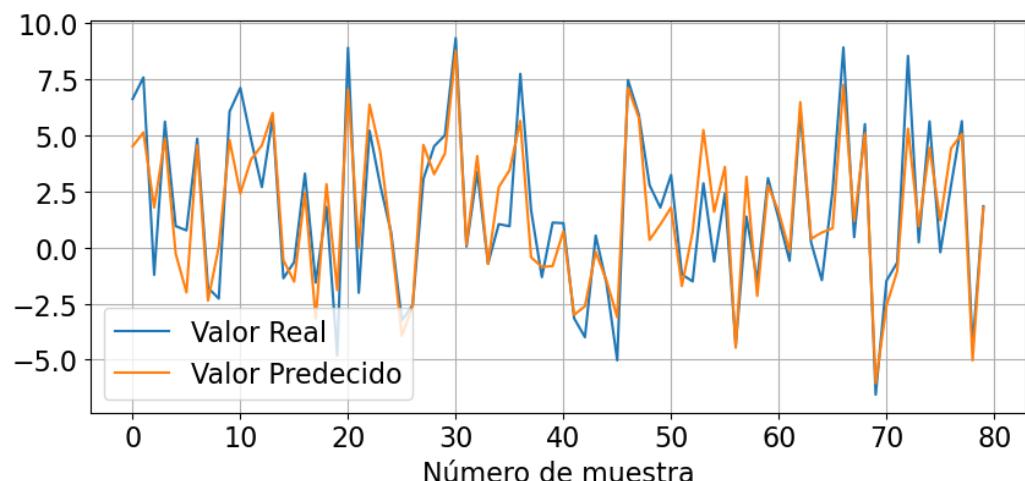


Fig. 39 – Resultados del ajuste de datos: Valores reales y valores predecidos



Descarga de los códigos de los ejercicios

Referencias

- Beale, M. H., Hagan, M., & Demuth, H. (2020). Deep Learning Toolbox™ User's Guide. In MathWorks. <https://la.mathworks.com/help/deeplearning/index.html>
- Demuth H, Beale M, Hagan M. (2018). Neural Network Toolbox™ User's Guide Neural network toolbox. MathWorks.
- Del Brío, M. B. y Molina S. (2007). Redes Neuronales y Sistemas Borrosos. 3ra edición. Ed. Alfaomega.
- Hagan, M. T., Demuth, H. B., Beale, M. H., & De Jesus, O. (2014). Neural Network Design 2nd Edition (2014). In Neural Networks in a Soft computing Framework.
- Lourakis, Manolis. (2005). A Brief Description of the Levenberg-Marquardt Algorithm Implemened by levmar
- Mishra, P. (2019). PyTorch Recipes. In PyTorch Recipes. <https://doi.org/10.1007/978-1-4842-4258-2>
- Pytorch, Biblioteca de aprendizaje automático Pytorch, 2021. <https://pytorch.org/>
- Scikit-learn, biblioteca de software de aprendizaje automático para Python, 2021. <https://scikit-learn.org/stable/>
- The Mathworks, Inc. MATLAB, Version 9.6, 2019. (2019). MATLAB 2019b - MathWorks. In www.Mathworks.Com/Products/Matlab.

Capítulo III - Redes de funciones de base radial (RBF)

Se realiza una introducción a las redes RBF, son una alternativa atractiva debido a su menor complejidad computacional y debido a su estrecha relación con los métodos bayesianos. Se presenta la estructura de estas redes y sus funciones, se realiza un análisis de errores locales y globales y se explica el procedimiento de aprendizaje con diferentes métodos. Por último, se resuelven ejercicios de compuerta XOR con datos simples y ruidosos, ajuste de datos y clasificación

Introducción

Las funciones de base radial presentan la particularidad que disminuye su respuesta al aumentar la distancia relativa respecto a un punto central. El centro, la escala de distancias y la forma precisa de la función radial son los parámetros del modelo. Las funciones radiales se pueden emplear en cualquier tipo de modelo (lineal o no lineal) y en cualquier tipo de red (de una sola capa o de varias capas). Una red RBF no es lineal si las funciones base pueden moverse o cambiar de tamaño o si hay más de una capa oculta (Diniz, 2020).

Las funciones RBF (en inglés Radial basis function) se introducen por Powell y otros colaboradores durante la década de 1980, utilizaron funciones de base radial para la interpolación exacta en un espacio multidimensional. Se requería que la función creada por la interpolación de base radial pasara exactamente a través de todos los objetivos en el conjunto de entrenamiento. Sin embargo, muchas aplicaciones tienen datos ruidosos, y la interpolación exacta generalmente da como resultado un sobreajuste de los datos de entrenamiento. Por tal motivo se utilizan redes con funciones de base radial robustas con interpolaciones suaves. No se intenta forzar la respuesta de la red para que coincida exactamente con las salidas de destino. Se pone énfasis en redes neuronales que deben generalizar correctamente ante situaciones cambiantes.

El proceso de aprendizaje de la red neuronal del tipo RBF consiste en encontrar una superficie en el espacio multidimensional que se ajuste mejor al conjunto de datos de entrenamiento. En particular, en el caso de las aplicaciones de comunicación, esta técnica resulta atractiva porque su aprendizaje permite la división de un espacio multidimensional en subregiones adecuadas donde encaja cada dato recibido. El RBF se ha propuesto como una alternativa atractiva debido a su menor complejidad computacional y debido a su estrecha relación con los métodos bayesianos. En el enfoque bayesiano, la decisión a favor de un símbolo se toma solo si la probabilidad de que el símbolo referido haya causado el vector de señal de entrada actual supera la probabilidad de que cualquier otro símbolo haya causado la misma entrada. El algoritmo RBF puede aproximarse a la solución bayesiana dentro de un tiempo de entrenamiento razonable, es un candidato potencial para ser empleado en una serie de aplicaciones donde se requieren filtros adaptativos no lineales (Del Brío, 2007).

Estructura de redes RBF

Las redes neuronales RBF son redes con dos capas. A diferencia de las redes Perceptrón, en la capa 1 de la red RBF, en lugar de realizar una operación de producto interno entre los pesos y la entrada, calculamos la distancia entre el vector de entrada y las filas de la matriz de pesos. Además, en lugar de agregar el sesgo, multiplicamos por el sesgo (Beale, 2020). Por lo tanto, la entrada neta para la neurona i en la primera capa se calcula mediante la siguiente ecuación:

$$n_i^1 = \|p - w_i^1\| \cdot b_i^1 \quad (3.1)$$

La matriz de pesos W contiene los centros de las funciones. Los sesgos b realiza un escalado de la función de transferencia. Muchas publicaciones utilizan la desviación estándar o varianza en lugar del término sesgo. Es simplemente una notación diferente, el funcionamiento de la red no se ve afectado. Para funciones de transferencia gaussianas, el sesgo está relacionado con la desviación estándar mediante la siguiente ecuación:

$$b = \frac{1}{\sigma \cdot \sqrt{2}} \quad (3.2)$$

En la primera capa de la red RBF, se utiliza mucho la función gaussiana. Se define con la siguiente ecuación:

$$\phi(r) = e^{-(\varepsilon \cdot r)^2} \quad (3.3)$$

Para redes neuronales la entrada de la función RBF la denominamos n , y la salida es a , entonces tenemos:

$$a = e^{-n^2} \quad (3.4)$$

Se muestra la respuesta de la función en la Fig. 40. Esta función es local, es decir que la salida tiende a cero para puntos lejanos al centro. Son distintas a las funciones sigmoides globales, cuya salida es cercana a 1 cuando la entrada neta tiende a infinito.

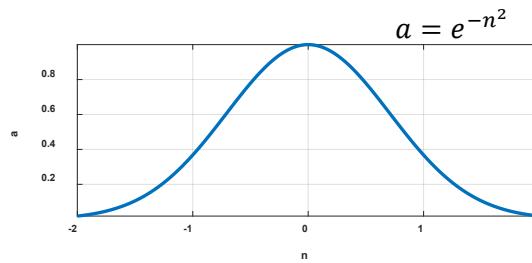


Fig. 40 – Función gaussiana

La segunda capa de la red RBF es una capa lineal estándar, definida por la siguiente ecuación

$$a^2 = W^2 \cdot a^1 + b^2 \quad (3.5)$$

En la Fig. 41 se muestra la estructura de la red RBF.

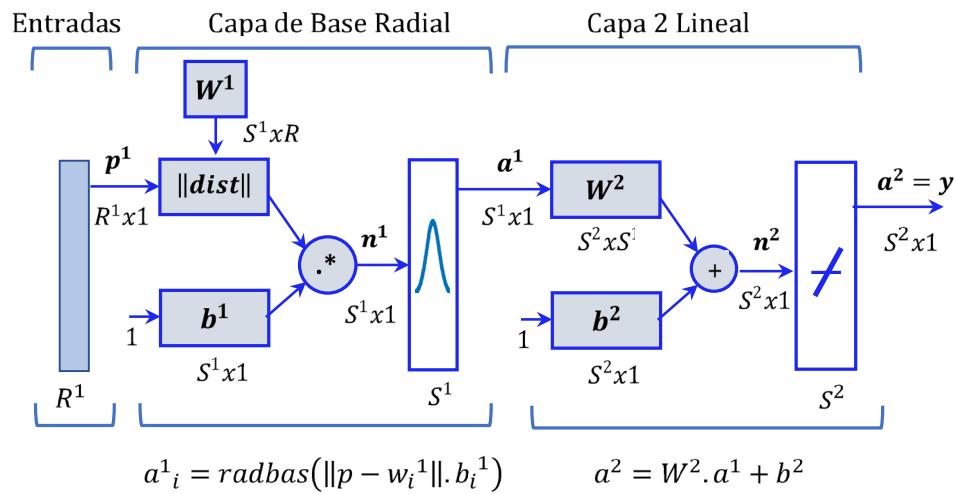


Fig. 41 – Red neuronal RBF

Funciones RBF

Las funciones de base radial, conocidas como RBF, son funciones reales que dependen de la distancia al origen (Demuth, 2018)

$$\phi(r) = \Phi(\|x\|) \quad (3.6)$$

O a algún centro x_k .

$$\phi(r) = \Phi(\|x - x_k\|) \quad (3.7)$$

La norma es $r = \|x\|$, generalmente se utiliza la norma euclíadiana.

La función radial es $\Phi: \mathbb{R}^s \rightarrow \mathbb{R}$, siendo $\Phi(\|x\|) = \phi(r)$.

La función RBF resulta constante para puntos que tienen la misma distancia al origen o a algún centro x_k , en este caso la función resulta radialmente simétrica respecto del centro correspondiente.

A continuación, se presentan distintas funciones RBF

- Función de distancia

$$\phi(r) = r \quad (3.8)$$

Esta función tiene asociada la matriz de distancia euclíadiana, con entradas

$$A_{j,k} = \|x_j - x_k\| \quad (3.9)$$

- Función Gaussiana

$$\phi(r) = e^{-(\varepsilon \cdot r)^2} \quad (3.10)$$

- Función multi cuadrática

$$\phi(r) = \sqrt{1 + (\varepsilon \cdot r)^2} \quad (3.11)$$

- Función multi cuadrática inversa (inversa de la función anterior)

$$\phi(r) = \frac{1}{\sqrt{1 + (\varepsilon \cdot r)^2}} \quad (3.12)$$

- Función Spline poliarmónico

$$\phi(r) = r^k, \quad k = 1, 3, 5, \dots \quad (3.13)$$

- Función Spline de capa delgada

$$\phi(r) = r^2 \cdot \ln(r) \quad (3.14)$$

El parámetro ε determina el decaimiento de las funciones a medida que se alejan las mediciones del centro RBF.

Análisis locales y globales

Las redes Perceptron multi capas utilizan funciones de transferencia global, y las redes RBF utilizan funciones de transferencia locales. En la red Perceptron todas las funciones de transferencia se superponen y aportan en forma significativa. Es decir, para cualquier valor de entrada, muchas funciones sigmoideas en la primera capa tendrán salidas significativas. Luego, en la segunda capa tienen que sumar o cancelar para producir la respuesta adecuada. En cambio, en la red RBF cada función solo está activa en un pequeño rango de la entrada. Para cualquier entrada dada, solo unas pocas funciones básicas estarán activas (Pytorch, 2021).

El enfoque global tiende a requerir menos neuronas en la capa oculta, ya que cada neurona contribuye a la respuesta en una gran parte del espacio de entrada. Sin embargo, para la red RBF, se requieren muchos centros locales, donde cada centro solo tiene un pequeño aporte a la salida de la red. Esto genera una alta dimensionalidad. Además, si se tienen muchas neuronas y parámetros se sobreajustan los datos de entrenamiento, por lo tanto, no se generaliza bien ante nuevas entradas.

Como ventaja del enfoque local, generalmente obtenemos un entrenamiento rápido, con algoritmos de solo dos etapas. El enfoque local puede ser muy útil para el entrenamiento adaptativo, en el que la red continúa entrenándose de forma incremental mientras se utiliza, como en los filtros adaptativos.

Procedimiento de aprendizaje de las redes RBF

Las redes RBF se pueden entrenar con diferentes algoritmos. Resulta distinto a las redes Perceptron multi capa, que casi siempre se entrena mediante algún algoritmo basado en gradientes, como por ejemplo descenso más pronunciado, o Levenberg-Marquardt (The Mathworks, 2019).

Las redes RBF se pueden entrenar utilizando algoritmos basados en gradientes. Sin embargo, debido a la naturaleza local de la función de transferencia y los centros y sesgos, tiende a haber muchos más mínimos locales en las superficies de error de las redes RBF respecto de las Perceptron multi capa. Por esta razón, los algoritmos basados en gradientes muchas veces no son satisfactorios para el entrenamiento de redes RBF. A veces se utilizan los algoritmos de gradientes para ajustar la red después de haberla entrenado inicialmente con algún otro método.

Los algoritmos de entrenamiento RBF más utilizados tienen dos etapas, que tratan las dos capas por separado. Los algoritmos difieren en cómo se seleccionan los pesos y sesgos de la primera capa. Los pesos de la segunda capa se pueden calcular utilizando un algoritmo lineal de mínimos cuadrados.

Un método simple para la primera capa es armar un patrón de cuadrícula en todo el rango de entrada para asignar centros en cada cuadrícula y elegir un sesgo constante. Este procedimiento no es óptimo, ya que hay zonas que requieren mayor complejidad de análisis. Además, muchas veces no se utiliza el rango completo del espacio de entrada, desperdiando recursos. Con este método de cuadrículas tenemos problemas de dimensionalidad a medida que los datos crecen.

Otro método es seleccionar algún subconjunto aleatorio de los vectores de entrada del conjunto de entrenamiento, en base a este subconjunto se seleccionan los centros. Esto asegura que los centros corresponden a datos de entrada indispensables para la red. Sin embargo, tenemos aleatoriedad en la selección, pudiendo tener problemas ya que no se abarcan todos los datos. Un enfoque más eficiente y elaborado es utilizar la capa competitiva de Kohonen, para agrupar el espacio de entrada. Los centros de grupos se convierten entonces en centros de función básica. Esto asegura que las funciones básicas se coloquen en regiones con actividad significativa.

Se enuncian distintos algoritmos de entrenamiento de redes neuronales RBF

- Mínimos cuadrados lineales
- Mínimos cuadrados ortogonales
- Agrupamiento (Clustering)
- Optimizaciones no lineales

A continuación, se explican algunos métodos

Mínimos cuadrados lineales en RBF (linear Least squares: LLS)

En este método LLS, en primer lugar, hay que fijar los pesos y sesgos de la primera capa de la red RBF. Esto se puede hacer fijando los centros en cuadrículas, o seleccionando aleatoriamente los centros de los vectores de entrada correspondientes al conjunto de datos de entrenamiento, también se puede usar algún método de agrupamiento (Demuth, 2018).

Luego los sesgos se pueden calcular utilizando la siguiente fórmula:

$$b_i^1 = \frac{\sqrt{S^1}}{d_{max}} \quad (3.15)$$

d_{max} es la distancia máxima entre los centros vecinos

Una vez fijados los parámetros de la primera capa, la segunda capa se entrena como una red lineal Adaline. Dado el conjunto de entrenamiento:

$$[p_1, t_1]; [p_2, t_2]; [p_3, t_3]; \dots [p_P, t_P]$$

Siendo p_i las entradas y t_i las salidas deseadas, para la primera capa tenemos las siguientes ecuaciones

$$a^1_p = radbas(n_p^1) \quad (3.16)$$

$$n_{i,p}^1 = \|p_p - w_i^1\| \cdot b_i^1 \quad (3.17)$$

Las salidas de la primera capa se conectan con la segunda. Entonces, en la segunda capa tenemos:

$$[a^1_1, t_1]; [a^1_2, t_2]; [a^1_3, t_3]; \dots [a^1_P, t_P] \quad (3.18)$$

$$a^2 = W^2 \cdot a^1 + b^2 \quad (3.19)$$

Mediante LLS se busca ajustar los pesos y sesgos, de manera de reducir el error cuadrático medio.

$$F(x) = \sum_{p=1}^P (t_p - a^2_p)^T \cdot (t_p - a^2_p) \quad (3.20)$$

La solución a este problema de mínimos cuadrados lineales es la solución de la red lineal Adaline.

Para simplificar la discusión, asumiremos un objetivo escalar y agruparemos todos los parámetros que estamos ajustando, incluido el sesgo, en un vector:

$$x = \begin{bmatrix} w_1^2 \\ b^2 \end{bmatrix} \quad (3.21)$$

De manera similar, incluimos la entrada de sesgo "1" como un componente del vector de entrada:

$$z_p = \begin{bmatrix} a^1_p \\ 1 \end{bmatrix} \quad (3.22)$$

Ahora la salida de la red, que normalmente escribimos en la forma:

$$a^2_p = (w_1^2)^T \cdot a^1_p + b^2 \quad (3.23)$$

Se puede escribir como:

$$a_p = x^T \cdot z_p \quad (3.24)$$

Esto nos permite escribir convenientemente una expresión para el error de suma cuadrada:

$$F(x) = \sum_{p=1}^P (e_p)^2 = \sum_{p=1}^P (t_p - a_p)^2 = \sum_{p=1}^P (t_p - x^T \cdot z_p)^2 \quad (3.25)$$

Se puede expresar mediante matrices de la siguiente manera:

$$t = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_p \end{bmatrix}; \quad U = \begin{bmatrix} u_1^T \\ u_2^T \\ \vdots \\ u_p^T \end{bmatrix} = \begin{bmatrix} z_1^T \\ z_2^T \\ \vdots \\ z_p^T \end{bmatrix}; \quad e = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_p \end{bmatrix} \quad (3.26)$$

El error lo expresamos con la siguiente ecuación:

$$e = t - U \cdot x \quad (3.27)$$

Y el índice de performance resulta

$$F(x) = (t - U \cdot x)^T \cdot (t - U \cdot x) \quad (3.28)$$

Para evitar el sobreajuste, modificamos esta ecuación en la siguiente ($\rho = \alpha/\beta$)

$$F(x) = (t - U \cdot x)^T \cdot (t - U \cdot x) + \rho \cdot \sum_{i=1}^n x_i^2 = (t - U \cdot x)^T \cdot (t - U \cdot x) + \rho \cdot x^T \cdot x \quad (3.29)$$

$$F(x) = t^T \cdot t - 2 \cdot t^T \cdot U \cdot x + x^T \cdot U^T \cdot U \cdot x + \rho \cdot x^T \cdot x \quad (3.30)$$

$$F(x) = t^T \cdot t - 2 \cdot t^T \cdot U \cdot x + x^T \cdot [U^T \cdot U + \rho \cdot I] \cdot x \quad (3.31)$$

Corresponde a la siguiente forma cuadrática:

$$F(x) = c + d^T \cdot x + \frac{1}{2} x^T \cdot A \cdot x \quad (3.32)$$

Siendo: $c = t^T \cdot t$; $d = -2 \cdot U^T \cdot t$; $A = 2 \cdot [U^T \cdot U + \rho \cdot I]$

Las características de la función cuadrática $F(x)$ dependen principalmente de la matriz de Hessiana (matriz A en nuestro caso). Por ejemplo, si los valores propios del Hessiano son todos positivos, entonces la función tendrá un mínimo global único. Se puede demostrar que esta matriz es positiva definida o positiva semi definida, es decir, nunca puede tener valores propios negativos.

Calculamos el gradiente de $F(x)$

$$\nabla F(x) = \nabla \left(c + d^T \cdot x + \frac{1}{2} x^T \cdot A \cdot x \right) = d + A \cdot x = -2 \cdot U^T \cdot t + 2 \cdot [U^T \cdot U + \rho \cdot I] \cdot x \quad (3.33)$$

Igualamos el gradiente de $F(x)$ para encontrar el punto de estacionariedad

$$-2 \cdot U^T \cdot t + 2 \cdot [U^T \cdot U + \rho \cdot I] \cdot x = 0 ; \quad [U^T \cdot U + \rho \cdot I] \cdot x = U^T \cdot t \quad (3.34)$$

Entonces el punto estacionario se obtiene con los valores óptimos de los parámetros x^*

$$[U^T \cdot U + \rho \cdot I] \cdot x^* = U^T \cdot t \quad (3.35)$$

Si la matriz de Hessiana solo tiene valores propios positivos, el índice de rendimiento $F(x)$ tendrá un mínimo global único. Los valores de los pesos y sesgos óptimos x^* se pueden calcular de la siguiente forma:

$$x^* = [U^T \cdot U + \rho \cdot I]^{-1} \cdot U^T \cdot t \quad (3.36)$$

Mínimos cuadrados ortogonales

Se puede utilizar mínimos cuadrados ortogonales para el entrenamiento de redes RBF. En este método se seleccionan subconjuntos para construir modelos lineales. Generalmente se eligen centros basados en todos los vectores de entrada. En diferentes pasos se agregan neuronas hasta que se cumplen algunos criterios. La selección de la nueva neurona se basa en cuánto se reducirá el error cuadrático medio al agregar la neurona. Normalmente, se busca maximizar la capacidad de generalización que tiene la red, colocando un número adecuado de neuronas.

Agrupamiento (Clustering) en RBF

Mediante este método se seleccionan los pesos y los sesgos en la primera capa de la red RBF. Se utilizan las redes competitivas mediante una capa competitiva de Kohonen y mediante Mapa de características autoorganizado (Self Organizing Feature Map). Consiste en agrupar los vectores de entrada del conjunto de entrenamiento. Después del entrenamiento, las filas de las redes competitivas contienen centros de grupos (o clústeres). Esto proporciona un enfoque para localizar centros y seleccionar sesgos para la primera capa de la red RBF. Si tomamos los vectores de entrada del conjunto de entrenamiento y realizamos una operación de agrupamiento en ellos, los prototipos resultantes (centros de grupos) se utilizan como centros para la red RBF. Además, se calcula la varianza de cada grupo individual y se usa para calcular los sesgos apropiados de cada neurona

Otro algoritmo de agrupamiento sencillo y muy utilizado es K-medias (K-means). En este método se realizan los siguientes pasos:

- Se elige un número de clases K.
- Se inicializan en forma aleatoria los centros de cada clase.
- Se asignan pertenencias a cada clase de la siguiente manera:

El patrón de entradas $X(n)$ pertenece al clúster i si:

$$\|X(n) - C_i\| < \|X(n) - C_s\| ; \forall s \neq i \text{ con } s = 1,2,3 \dots K \quad (3.37)$$

Siendo C_i y C_s los centros de las clases

Es decir, para cada entrada se le asigna la clase que tenga el centro más cercano.

Definimos P_i como factor de pertenencia al grupo:

$$P_i = \begin{cases} 1 & \text{si } \|X(n) - C_i\| < \|X(n) - C_s\| ; \text{ con } s = 1,2,3 \dots K, \text{ si pertenece al grupo} \\ 0 & \text{si no pertenece al grupo} \end{cases} \quad (3.38)$$

- Se modifican los centros de cada grupo calculando el valor medio de todas las entradas que pertenecen al grupo, se utiliza la siguiente fórmula:

$$C_i = \frac{1}{N} \cdot \sum_{n=1}^N P_i \cdot X(j) ; \quad j = 1,2,3 \dots p ; i = 1,2,3 \dots K \quad (3.39)$$

- Se repiten últimos pasos c) y d) hasta que los centros se mantengan constantes respecto del paso anterior. O hasta que el desplazamiento sea muy bajo.

$$\|C_i(t) - C_i(t-1)\| < \varepsilon ; \quad \forall i = 1,2,3 \dots K \quad (3.40)$$

Además de calcular los pesos de la primera capa, el proceso de agrupación puede proporcionar los sesgos de la primera capa. Para cada neurona, luego de calcular el centro, se calcula la distancia promedio entre el centro y sus vecinos.

$$d_i = \sqrt{\sum \|C_i - C_m\| \cdot \|C_i - C_s\|} \quad (3.41)$$

Siendo C_m y C_s los centros próximos a C_i . Se calculan los sesgos de la primera capa mediante las desviaciones de cada grupo mediante la siguiente ecuación:

$$b_i^{-1} = \frac{1}{\sqrt{2} \cdot d_i} \quad (3.42)$$

Por lo tanto, cuando un clúster es amplio, la función de base correspondiente también lo será. Tenga en cuenta que, en este caso, cada sesgo de la primera capa será diferente. Esto proporciona debería generar una red más eficiente respecto a las redes con sesgos iguales.

Una vez determinados los pesos y los sesgos de la primera capa, se utilizan mínimos cuadrados lineales para encontrar los pesos y los sesgos de la segunda capa.

En este método uno esperaría que los datos de entrenamiento estén ubicados en las regiones donde la red será más utilizada y, por lo tanto, la aproximación de funciones será más precisa. Los pesos resultantes representarán los centros de agrupación de los vectores de entrada del conjunto de entrenamiento. Esto asegurará que tendremos funciones de base ubicadas en áreas donde es más probable que ocurran los vectores de entrada.

Sin embargo, puede existir un inconveniente en el método de agrupamiento para redes RBF. El método solo tiene en cuenta la distribución de los vectores de entrada; no considera los objetivos. Es posible que la función que estamos tratando de aproximar sea más compleja en regiones para las que hay menos entradas. Para este caso, el método de agrupamiento no distribuirá los centros de manera adecuada. Además, se tiene información de las salidas que no se utiliza.

Optimizaciones no lineales

Se pueden utilizar métodos no lineales de gradiente descendente en las 2 capas de la red RBF, tal como se realiza en las redes Perceptron multi capas, donde todos los pesos y todos los sesgos se ajustan al mismo tiempo. Estos métodos no se utilizan generalmente para el entrenamiento completo de redes RBF, debido a que puede haber mínimos locales no resueltos en su superficie de error. Sin embargo, la optimización no lineal se puede utilizar para el ajuste fino de los parámetros de la red, después del entrenamiento inicial mediante uno de los métodos de dos etapas que presentamos anteriormente.

Los métodos de optimización no lineales fueron presentados anteriormente mediante la propagación hacia atrás de los errores y cálculo del gradiente en redes. Para redes RBF hay que tener en cuenta las ecuaciones correspondientes de la primera capa y el significado de los pesos y los sesgos (Tensorflow, 2021).

Ejercicios de redes RBF

Ejercicio 3.1

Red neuronal RBF para compuerta XOR – Analítico y Matlab

Se pide resolver la compuerta XOR, ver Fig. 42, con una red neuronal RBF simple

Compuerta XOR:

Categoría 1: $p_1^T = [-1 \ -1]$ respuesta: $t_1 = -1$; $p_4^T = [1 \ 1]$ respuesta: $t_4 = -1$

Categoría 2: $p_2^T = [-1 \ 1]$ respuesta: $t_2 = 1$; $p_3^T = [1 \ -1]$ respuesta: $t_3 = 1$

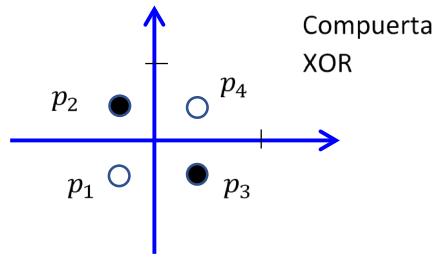


Fig. 42 – Compuerta XOR

Resolución

La compuerta XOR no es linealmente separable, se puede resolver con una red RBF.

Trabajaremos con el esquema de la Fig. 41. Planteamos como solución tener respuestas con valores altos para los puntos cercanos a p_2 y a p_3 y valores bajos a medida que se alejan los puntos. Entonces elegimos los centros p_2 y p_3 y con estos centros armamos la matriz W^1

$$W^1 = \begin{bmatrix} p_2^T \\ p_3^T \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \quad (3.43)$$

Para cada uno de los puntos p_i la distancia euclíadiana al origen es $\sqrt{2}$.

Para un sesgo $b^1 = \begin{bmatrix} 1,8 \\ 1,8 \end{bmatrix}$ calculamos las salidas de las funciones RBF cuando la entrada es $p^T = [0 \ 0]$, es decir en el origen

$$n_i^1 = \|p - w_i^1\| \cdot b_i^1$$

$$n_1^1 = \left\| \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\| \cdot 1,8 = \sqrt{2} \cdot 1,8 = 2,5456$$

$$a^1_i = radbas(\|p - w_i^1\| \cdot b_i^1) = radbas(a^1_i)$$

$$a^1_1 = a^1_2 = radbas(2,5456)$$

Utilizando función gaussiana $a = radbas(n) = e^{-n^2}$

$$a^1_1 = a^1_2 = radbas(2,5456) = e^{-2,5456^2} = 0,015$$

Siendo un valor bajo.

Ahora calculamos la salida de la función RBF para la entrada p_2 , es decir en el centro elegido

$$n_1^1 = \left\| \begin{bmatrix} -1 \\ 1 \end{bmatrix} - \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\| \cdot 1,8 = 0$$

$$a^1_1 = a^1_2 = radbas(0) = e^{-0^2} = 1$$

Se verifica el máximo de la función RBF para distancia 0.

Por último, elegimos $W^2 = [2 \ 2]$ y $b^2 = [-1]$ para la capa 2 lineal, con estos valores se adaptan las salidas a las respuestas deseadas -1 y +1. Aplicando la siguiente ecuación obtenemos la salida de la red neuronal RBF.

$$a^2 = W^2 \cdot a^1 + b^2 \quad (3.44)$$

Mediante el siguiente programa en Matlab®, se gráfica la respuesta de la red para distintas entradas

```
% Se definen los parámetros de la red
W1 = [-1 1; 1 -1];
b1 = [1;1];
b1 = [1.8;1.8];
W2 = [2 2];
b2 = [-1];
% Asignamos valores a la entrada p y calculamos a: salida de la red
p=[1 1]
a = W2(1)*exp(-(pdist2(p,W1(1,:)).*b1(1)).^2) + W2(2)*exp(-(pdist2(p,W1(2,:)).*b1(2)).^2) + b2
% Resultado correcto -1
% Se grafica la respuesta de la red para distintos valores de p
f=@(p) W2(1)*exp(-(pdist2(p,W1(1,:)).*b1(1)).^2) + W2(2)*exp(-(pdist2(p,W1(2,:)).*b1(2)).^2) + b2
x=-3:0.05:3; y=x;
[XX,YY] = meshgrid(x,y);
gg = [XX(:) YY(:)]';
z = f(gg)';
zz= reshape(z, length(XX),length(XX)) ;
mesh(XX,YY,zz) %surf
axis tight ; grid on
xlabel('p(1)') ; ylabel('p(2)') ; zlabel('Salida') ;
figure ; contour(XX,YY,zz); hold on
scatter([-1 1],[1 -1],40,'o', 'LineWidth',8 , 'MarkerEdgeColor','r')
scatter([-1 1],[-1 1],40,'x', 'LineWidth',8 , 'MarkerEdgeColor','b')
axis tight ; grid on ; % plot3(gg(1,:),gg(2,:),z)
```

En la Fig. 43 se muestran los resultados, a la izquierda se observa la correcta clasificación para las 4 entradas de la compuerta XOR p_1 , p_2 , p_3 y p_3 . También se observan los máximos para los centros elegidos. A la derecha se observan las 4 entradas y el gráfico de contorno con sus circunferencias.

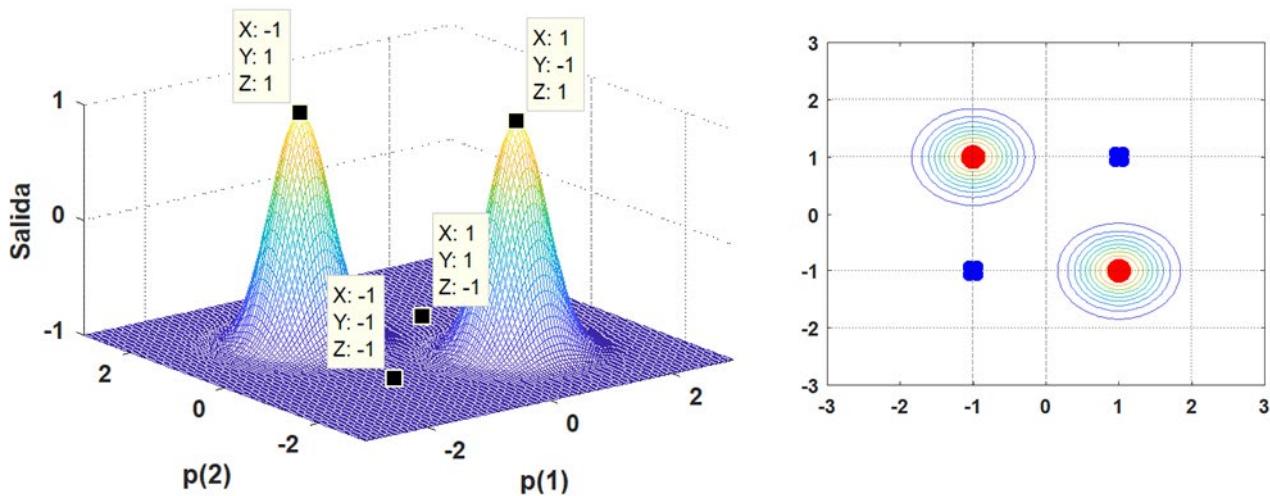


Fig. 43 – Resultados de la Red neuronal RBF para una compuerta XOR. Izquierda) gráfico mesh, derecha) gráfico de contorno

Ejercicio 3.2

Red neuronal RBF para compuerta XOR con datos ruidosos

Mediante Matlab® se pide resolver la compuerta XOR con una red neuronal RBF, para entradas con ruido

Resolución

```
%% Ejemplo de red neuronal RBF y problema XOR para entradas con ruido
N= 60; % Cantidad de muestras para cada grupo
d = 1; % Distancias
a=1.5 ;
% Definimos las entradas con 4 grupos y graficamos
Datos1 = [-d+ -a/2+a*rand(1,N) d+-a/2+a*rand(1,N); -a/2+a*rand(1,N)+d -a/2+a*rand(1,N)-d] ;
Datos2 = [d+-a/2+a*rand(1,N) -d+-a/2+a*rand(1,N); -a/2+a*rand(1,N)+d -a/2+a*rand(1,N)-d];
plot(Datos1(1,:),Datos1(2,:),'k+'); hold on
plot(Datos2(1,:),Datos2(2,:),'b*') ; grid on
% Definimos matriz P de entradas que contienen las 4 clases
P = [Datos1 Datos2];
% Definimos vector T de respuestas correctas, codificado con -1 y +1
T = [repmat(-1 ,1,length(Datos1)) repmat(1,1,length(Datos2))];
% Creamos la red RBF EXACTA y la mostramos
% net = newrbe(P,T,1);
net = newrb(P,T,0,1,4);
view(net)
% Generamos una grilla y la clasificamos para generar mapa de colores
xx = -2:0.02:2;
[XX,YY] = meshgrid(xx,xx);
gg = [XX(:) YY(:)]';
% Clasificamos los puntos de la grilla con la red neuronal y Graficamos
tt = sim(net,gg);
```

```

figure(1)
largo= length(xx);
m1 = mesh(XX,YY,reshape(-tt,largo,largo)-4);
m2 = mesh(XX,YY,reshape( tt,largo,largo)-4);
set(m1,'facecolor',[0.8 0.25 .75],'linestyle','none');
set(m2,'facecolor',[1 0.9 .6],'linestyle','none');
view(2)
% Graficamos los centros de la red RBF
scatter(net.iw{1}(:,1),net.IW{1}(:,2),70,'o', 'LineWidth',8 , 'MarkerEdgeColor','r')
% Simulamos los datos de entrenamiento y calculamos porcentaje de clasificaciones correctas
Y = net(P);
Porcentaje_correcto = 100 * length(find(T.*Y > 0)) / length(T)
cantidad_neuronas = net.layers{1}.size
% Graficamos las respuestas correctas y las respuestas de la red
figure;
plot(T,'r', 'linewidth',2) ; hold on; grid on
plot(Y,'b', 'linewidth',1)
xlabel('Número de muestra')
legend('Respuestas correctas','Respuestas de la red')

```

En la Fig. 44 se observa la configuración de la red y la correcta clasificación de las entradas. Cada región del gráfico corresponde a la categoría correspondiente (-1 o +1). También se observan los 4 centros de la red RBF.

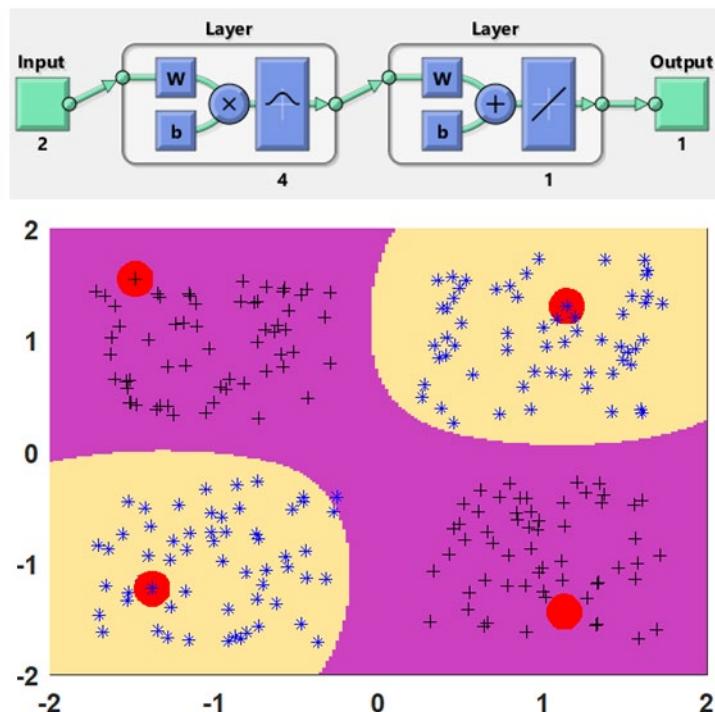


Fig. 44 – Resultados de la Red neuronal RBF con 4 centros para compuerta XOR con entradas ruidosas

Si en el programa anterior utilizamos una red neuronal de base radial exacta, y además no limitamos el número máximo de neuronas, obtendremos una red neuronal con 240 neuronas en la primera capa, que coincide con el número de entradas. Es decir que tenemos 240 centros indicados con círculos en la Fig. 45.

Entonces, en el programa anterior solamente modificamos estas 2 líneas de la siguiente forma:

```
net = newrbe(P,T,1);
%net = newrb(P,T,0,1,4);
```

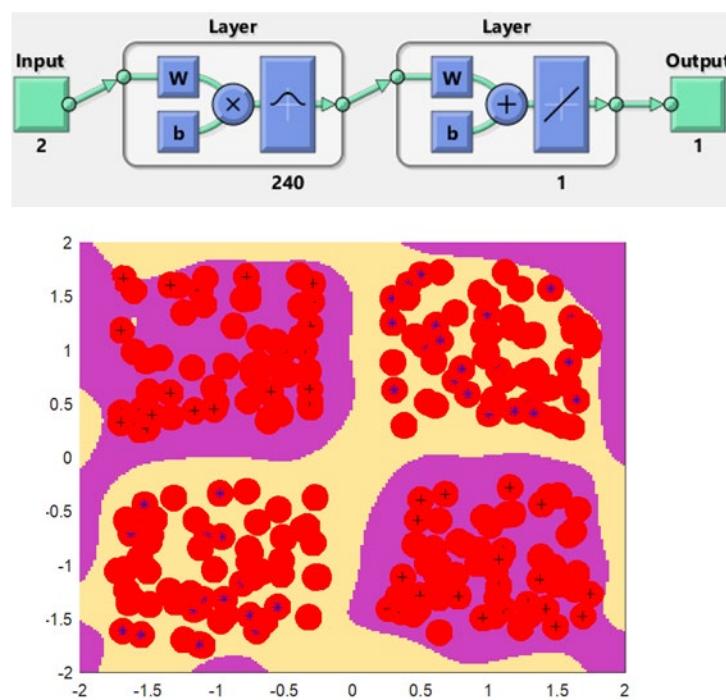


Fig. 45 – Resultados de la Red neuronal RBF con 240 centros para compuerta XOR con entradas ruidosas

Ejercicio 3.3

Ejercicio básico de ajuste de datos Red neuronal RBF

Dado un conjunto de datos de entrenamiento, se pide crear una red neuronal RBF y probar su funcionamiento

Resolución

% Datos de entrenamiento

X = -1:.1:1;

T=[-1.9204 -1.154 -0.1458 0.7542 1.281 1.32 0.9218 0.2672 -0.4026 -0.8688 -1 ...

-0.986 -0.5294 -0.0024 0.4144 0.592 0.4898 0.1632 -0.2624 -0.6378 -0.8402];

% Graficamos los datos de entrenamiento

plot(X,T,'x');

```

title('Vector de entrenamiento');
ylabel('Vector de salidas deseadas T');
xlabel('Vector de entradas P');
% Entrenamos la red RBF
eg = 0.01; % suma de error cuadrático esperado
sc = 1; %constante de dispersión, si es mayor, tenemos aproximación suave
net = newrb(X,T,eg,sc);
view(net)
% Graficamos resultados
figure; plot(X,T,'x', 'markersize',12); hold on; xlabel('Entradas');
% Clasificamos datos y graficamos
X = -1: .01: 1 ;
Y = net( X );
plot(X,Y);legend({'Salida de la red neuronal','Salida deseada'})
% Graficamos algunas funciones de transferencia
% Para entender el funcionamiento y la respuesta de la red RBF,
% se calcula y se grafica una suma ponderada de todas las funciones RBF
% Graficamos la función de transferencia a = radbas(n) = exp(-n^2)
x = -4:.05:4;
a = radbas(x);
figure; plot(x,a,'b') ; title('Función de transferencia RBF');
xlabel('Entrada p'); ylabel('Salida a');
% Y graficamos la suma ponderada de distintas funciones RBF
a2 = radbas(x+1.5); a3 = radbas(x-2);
a4 = a + a2 + a3 *0.55;
figure; plot(x,a,'b',x,a2,'r--',x,a3,'g--',x,a4,'m')
title('Suma ponderada de las funciones RBF');
xlabel('Entrada p'); ylabel('Salida a');

```

En la Fig. 46 se muestra la salida deseada en función de las entradas y la salida de la red neuronal

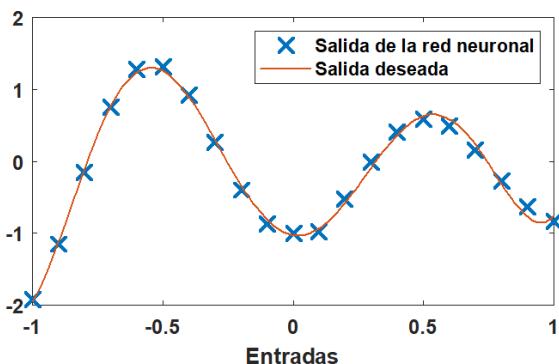


Fig. 46 – Salida deseada y salida de la red neuronal

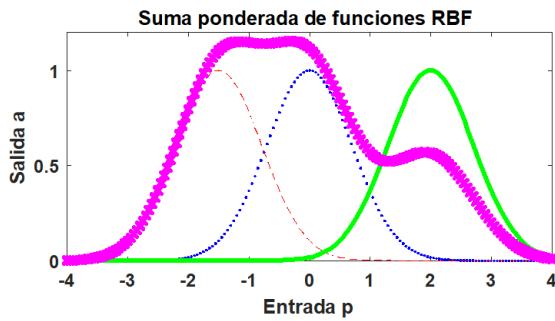


Fig. 47 – Ejemplo de suma ponderada de funciones RBF

Ejercicio 3.4

Red neuronal RBF para cálculos de regresión con Python

Mediante el módulo Keras de Tensorflow se genera una red neuronal RBF para ajustes de datos. Se utilizan los siguientes comandos:

```
model = Sequential()
capa_rbf = RBF_Capa(12, initializer=inicio_centros , betas=2.0, input_shape=(1,))
outputlayer = Dense(1, use_bias=False)
model.add(capa_rbf)
model.add(outputlayer)
```

Donde RBF_Capa genera la capa RBF de la red neuronal con 12 neuronas. En la Fig. 48 se muestran los datos originales y los datos predecidos con la red. Se obtiene un error cuadrático medio bajo (MSE: 0,031).

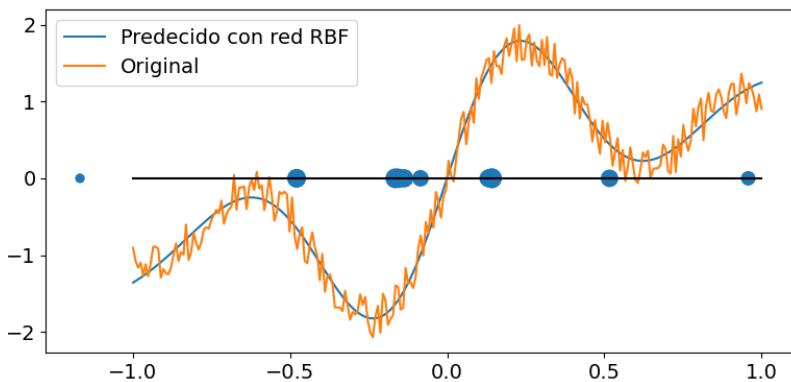


Fig. 48 – Ejemplo de red RBF para ajuste de datos, los círculos representan los centros de las neuronas RBF



Descarga del código de este ejercicio

Ejercicio 3.5

Red neuronal RBF para clasificar datos en 2 dimensiones con Python

Se genera una red neuronal RBF para ajustes de datos. Se utilizan los siguientes comandos:

```
model = Sequential()
capa_rbf = RBF_Capa(15, initializer=inicio_centros , betas=2.0, input_shape=(2,))
###input_shape=(1,)
outputlayer = Dense(5, use_bias=False)
model.add(capa_rbf)
model.add(outputlayer)
model.add(Dense(5, activation='softmax'))
```

Donde RBF_Capa genera la capa RBF de la red neuronal con 15 neuronas. En la Fig. 49 se muestran los datos originales y los datos predecidos con la red. Se observan los 15 centros de las neuronas RBF. Se obtiene una precisión del 98 %.

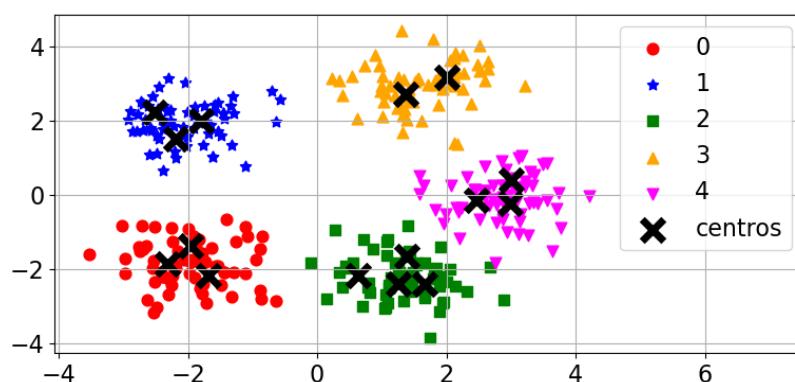


Fig. 49 – Ejemplo de red RBF para clasificación de datos



Descarga de los códigos de los ejercicios

Referencias

- Beale, M. H., Hagan, M., & Demuth, H. (2020). Deep Learning Toolbox™ User's Guide. In MathWorks. <https://la.mathworks.com/help/deeplearning/index.html>
- Demuth H, Beale M, Hagan M. (2018). Neural Network Toolbox™ User's Guide Neural network toolbox. MathWorks.
- Diniz, P. S. R. (2020). Adaptive Filtering. In Adaptive Filtering. Springer International Publishing. <https://doi.org/10.1007/978-3-030-29057-3>
- Del Brío, M, B. y Molina S. (2007). Redes Neuronales y Sistemas Borrosos. 3ra edición. Ed. Alfaomega.
- Pytorch, Biblioteca de aprendizaje automático Pytorch, 2021. <https://pytorch.org/>
- Tensorflow, Biblioteca de aprendizaje automático Tensorflow, desarrollada por Google, 2021, <https://www.tensorflow.org/>
- The Mathworks, Inc. MATLAB, Version 9.6, 2019. (2019). MATLAB 2019b - MathWorks. In www.Mathworks.Com/Products/Matlab.

Capítulo IV - Reconocimiento estadístico de patrones, aprendizaje automático y redes neuronales

El reconocimiento de patrones se ocupa de estudiar, analizar y procesar datos de procesos científicos tecnológicos relacionados a objetos físicos o abstractos, para extraer información de grupos o clases de dichos objetos.

En este capítulo se presentan diferentes técnicas de reconocimiento estadístico de patrones y redes neuronales. Se explica el aprendizaje y la capacidad de generalización de los modelos. Se explican diferentes métodos de validación, matriz de confusión y métricas que son necesarias para evaluar correctamente los modelos. Se muestran ejercicios analíticos y numéricos de K-NN (K vecinos más cercanos), Análisis de Componentes Principales PCA, redes neuronales, funciones de distribución, etc. Resulta muy importante conocer diferentes técnicas, para aplicar el método más adecuado, reducir errores y tener mayor generalización.

Reconocimiento de patrones con redes neuronales

Al aprendizaje automático, muchas personas lo definen como "el proceso que imita el cerebro humano". Hay que entender que un algoritmo muchas veces imita al cerebro humano, con cierta inteligencia, pero trabaja en forma distinta. En muchas aplicaciones se pueden resolver problemas complejos y obtener mejores resultados. Y en otras aplicaciones puede ser muy difícil de implementar, o puede fallar el algoritmo debido a que no se tienen los datos suficientes de entrenamiento o existen variables difíciles de medir. Muchas variables o percepciones humanas pueden ser detectadas fácilmente por una persona, por ejemplo, un diagnóstico médico, una radiografía o un análisis del comportamiento social. Con programas entrenados correctamente se pueden obtener mejores resultados que los humanos, sin embargo, hay que utilizar con cuidado los resultados obtenidos ya que los sistemas pueden fallar (Bishop, 2006).

Descripción del reconocimiento de patrones

Un patrón es una regularidad en un conjunto de datos o en determinadas nociones abstractas. El reconocimiento de patrones es un proceso para encontrar regularidades y similitudes en los datos utilizando mediciones o datos de aprendizaje. Estas similitudes se pueden encontrar en base a análisis estadísticos, datos históricos o mediante algoritmos.

En el reconocimiento de patrones, primero hay que recopilar datos. Los datos deben filtrarse y procesarse previamente para que el sistema pueda extraer las características de estos. Según el tipo de sistema de datos, se elegirá el algoritmo apropiado que puede ser de Clasificación, Agrupación o de Regresión para reconocer patrones (Diniz, 2013).

A continuación, detallamos distintos algoritmos para el reconocimiento de patrones (Webb, 2011).

- Clasificación supervisada: estos algoritmos clasifican objetos nuevos basándose en la información de muestras ya clasificadas, es decir se entrena con datos que contienen entradas y respuestas o clases (grupos de pertenencia). En la clasificación, el algoritmo asigna etiquetas a los datos según las características predefinidas. Son parte del aprendizaje supervisado.
- Clasificación no supervisada: a partir de una muestra no clasificada busca la clasificación de esta. En el entrenamiento no se conocen las respuestas.
- Agrupamiento (clustering). Un algoritmo divide los datos en varios grupos en función de la similitud de características. Este aprendizaje se considera no supervisado. A veces tenemos que analizar datos que son muy complicados y no existe una forma obvia de clasificarlos en diferentes categorías. Las redes neuronales se pueden utilizar para identificar características especiales de estos datos y clasificarlos en diferentes categorías sin conocimiento previo de los datos. Esta técnica es útil en la minería de datos para usos comerciales y científicos.
- Regresión. Los algoritmos de regresión intentan encontrar una relación entre las variables de entrada y de salida. Para luego predecir variables dependientes desconocidas basándose en las relaciones encontradas. Se basa en el aprendizaje supervisado.
- Predicción. Si tenemos una red que funciona correctamente para modelar una secuencia conocida de valores, se pueden predecir valores a futuro. Un ejemplo muy utilizado es la predicción del mercado de valores.
- Asociación: las redes neuronales pueden ser entrenadas para "recordar" una serie de patrones, de modo que cuando se presenta una versión distorsionada de un patrón en particular, la red la asocie con el más cercano en su memoria y devuelva la versión original de ese patrón en particular.

La clasificación, en un sentido matemático, implica dividir un espacio de n dimensiones en varias regiones, y dado un punto en el espacio, debe decir a qué región pertenece. Esta idea se utiliza en muchas aplicaciones del mundo real, por ejemplo, en varios programas de reconocimiento de patrones. Cada patrón se transforma en un punto multidimensional y se clasifica en un grupo determinado, cada uno de los cuales representa un patrón conocido (Goodner, 2001).

Se debe tener en cuenta la correcta selección de variables, es decir hay que determinar cuál es el conjunto de características o variables más adecuado para describir y analizar los objetos de estudio (Gray, 2004).

El reconocimiento de patrones se puede realizar mediante redes neuronales o puede ser estadístico (Reconocimiento estadístico de patrones: REP).

A continuación, se trabaja con redes neuronales en el entorno Matlab®.

Ajuste de datos o Regresión

En los problemas de ajustes de datos se busca que la red neuronal encuentre las salidas numéricas de un conjunto de entradas con datos numéricos. La herramienta de ajuste de datos (nftool) resuelve el problema de ajuste de datos de entrada-salida con una red neuronal de alimentación directa (“feedforward”) de dos capas entrenada con algoritmos Levenberg-Marquardt, gradiente conjugado o con método Bayesiano. Mediante esta herramienta se pueden cargar datos propios, o importar un conjunto de datos de Matlab®.

Reconocimiento de Patrones - Clasificación

En los problemas de reconocimiento de patrones con Clasificación se busca que la red neuronal clasifique las entradas en un conjunto de categorías de salida. Es decir que se busca hallar la categoría correspondiente a la entrada analizada. En Matlab® se puede utilizar la herramienta nprtool, o escribir código utilizando la siguiente función:

```
net = patternnet( hiddenSizes, trainFcn, performFcn )
```

Agrupamiento (clustering)

En los problemas de agrupación, se desea que una red neuronal agrupe los datos según su similitud. Se pueden utilizar redes con mapas autoorganizados (SOM: Self Organizing Map). Estas redes consisten en una capa competitiva capaz de clasificar un conjunto de datos de vectores con cualquier número de dimensión. La cantidad de clases que puede clasificar está dada por la cantidad de neuronas que tiene la capa. Las neuronas están organizadas en una topología 2D, lo que permite que la capa forme una representación de la distribución y una aproximación bidimensional de la topología del conjunto de datos.

En Matlab® se puede utilizar la herramienta nctool.

En la Tabla VI presentamos diferentes herramientas de redes neuronales en Matlab® (The Mathworks, 2019). En la tabla también se muestran los comandos necesarios para iniciar estas herramientas que contienen interfaces gráficas de usuario (GUI).

Tabla VI – Herramientas gráficas de Matlab® para redes neuronales

Comando Matlab® y nombre	Descripción
nnstart “Neural Network Start”	Abre la interfaz de Redes y Manejo de Datos, que permite importar, crear, usar y exportar redes neuronales y datos. Dispone de una ventana con comandos de inicio para herramientas de ajuste de datos con redes neuronales (nftool), reconocimiento de patrones (nprtool), agrupamiento (nctool) y series de tiempo (ntstool). También proporciona enlaces a listas de conjuntos de datos, ejemplos y otra información útil de ayuda introductoria.
nftool “Neural fitting”	Herramienta para la resolución de un problema de ajuste de datos, resolviéndolo con una red feedforward de dos capas entrenada con los algoritmos Levenberg-Marquardt, gradiente conjugado o con método Bayesiano.
nprtool “Neural Pattern Recognition”	Herramienta para reconocimiento de patrones. Sirve de guía para resolver un problema de clasificación utilizando una red de alimentación directa (feedforward) de dos capas con neurona de salida sigmoidea
nctool “Neural Clustering”	Herramienta para la resolución de un problema de agrupamiento utilizando un mapa autoorganizado. El mapa forma una representación comprimida del espacio de entrada, que refleja tanto la densidad relativa de los vectores de entrada en ese espacio, como una representación comprimida bidimensional de la topología del espacio de entrada.
ntstool “Neural Time Series”	Herramienta gráfica utilizada para el análisis de series de tiempo con redes neuronales, con ejemplo de ajustes de datos utilizando una red feedforward de dos capas

En la Fig. 50 se muestra la herramienta “Neural Network Start” del software Matlab®, se abre con el comando nnstart y dispone de accesos hacia las otras herramientas.

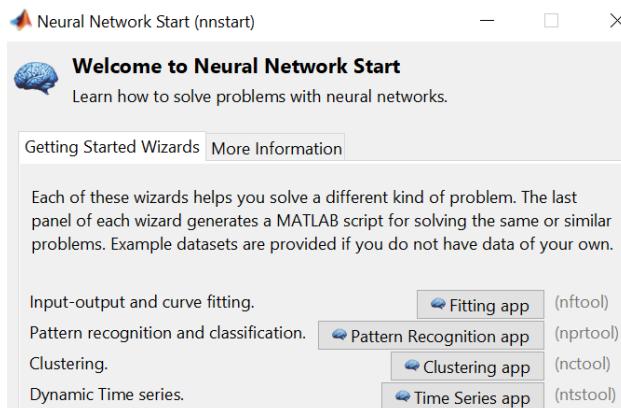


Fig. 50 – Pantalla principal de “Neural Network Start” del software Matlab®

Datos de entrenamiento, de validación y de testeo

Mediante los datos de entrenamiento las muestras se utilizan para ajustar el modelo (sesgos y pesos en el caso de una red neuronal). Es decir que el modelo aprende con los datos de entrenamiento (Tensorflow, 2021).

El conjunto de datos de validación se utiliza para medir la generalización de la red y para detener el entrenamiento cuando deja de mejorar la generalización del modelo. Estos datos proporcionan una evaluación de la respuesta del modelo mientras se entrena con el conjunto de datos de entrenamiento. Entonces, el conjunto de datos de validación puede afectar al modelo, pero solo indirectamente. Es decir que el conjunto de datos ayuda durante la etapa de desarrollo del modelo.

El conjunto de datos de testeo o prueba no tiene ningún efecto sobre el entrenamiento y, por lo tanto, proporcionan una medida del rendimiento de la red independiente del entrenamiento. Se utiliza una vez que el modelo está entrenado. Es decir, que la red primero se debe entrena con los datos de entrenamiento y de validación. A veces, el conjunto de validación se utiliza como conjunto de testeo o prueba, pero no es recomendable. El conjunto de testeo o prueba generalmente se debe seleccionar para que contenga datos que abarquen todas las clases posibles, para que luego la red trabaje correctamente en el mundo real (Subasi, 2020).

En la Fig. 51 se muestra un ejemplo de selección de porcentajes de datos de entrenamiento, datos de validación y datos de testeo. Estos porcentajes pueden variar y no existe un criterio universal, en la bibliografía se encuentran distintas recomendaciones (Scikit-learn, 2021).

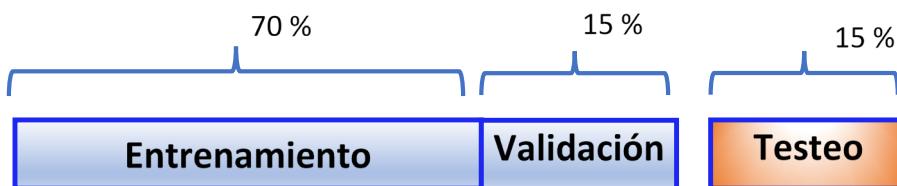


Fig. 51 – Ejemplo de porcentajes de datos de entrenamiento, validación y testeo

Ejercicios

Ejercicio 4.1

Ejemplos de redes neuronales en Matlab® para ajuste de datos, agrupamiento, reconocimiento de patrones y series temporales

Se utiliza la herramienta de Matlab®: “Neural Network Start”. Mediante el comando nnstart, abrir la interfaz gráfica de usuario (GUI). Dispone de ejemplos y accesos directos hacia las otras herramientas.

Se pide analizar el funcionamiento para:

- Ajuste de datos
- Agrupamiento
- Reconocimiento de Patrones
- Series temporales dinámicas

Ejercicio 4.2

Ajuste de datos con redes neuronales estáticas mediante herramienta nftool de Matlab

Se pide:

- Abrir la herramienta de ajustes de datos de Matlab® (nftool), al cargar datos, seleccionar “simple fitting problem”, genera datos en forma aleatoria (mediante la herramienta “simplefit_create”). Esta herramienta permite acceder a datos de ejemplo de Matlab® o cargar datos propios. Posteriormente seleccionar los porcentajes de entrenamiento, validación y testeо, por ejemplo 70%, 15% y 15% respectivamente. Asignar 10 neuronas para la capa oculta. Seleccionar algoritmo Levenberg-Marquardt y pulsar el comando correspondiente para “entrenar” la red. Presionar los comandos “plot fit” y “plot regression” para mostrar los resultados del entrenamiento y del testeо de la red. En la Fig. 52 se muestra la estructura de la red, se utilizan entradas y salidas analógicas. También se observa el ajuste de datos con errores bajos y todos los coeficientes de regresión R son cercanos a 1.
- Luego, en la pantalla final generar el código de programa (script) correspondiente, mediante el comando “Advanced Script”.
- Modificar la cantidad de neuronas que contiene la red y analizar las respuestas obtenidas.

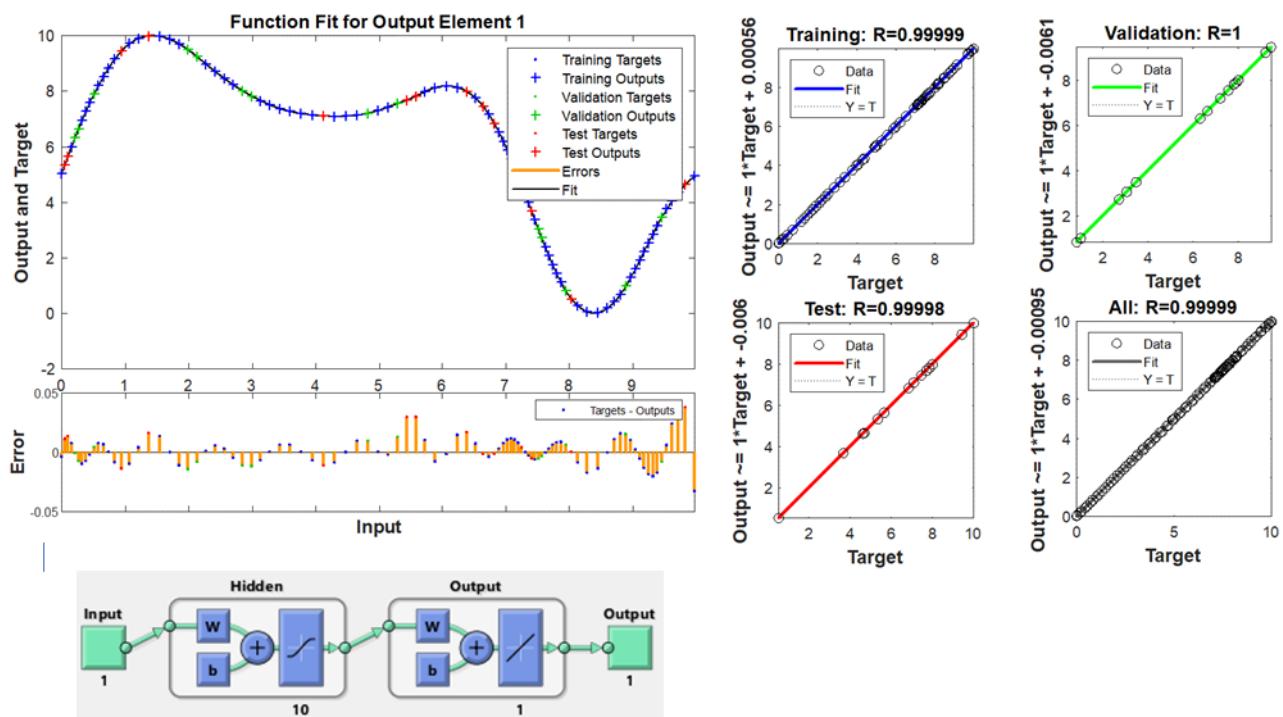


Fig. 52 – Resultados del Ajuste de datos con Matlab®

A continuación, se muestra el programa modificado para una mejor comprensión, a su vez se modifican las primeras líneas para importar los datos correspondientes. Este script resulta independiente de la herramienta nftool.

```

%% Ejemplo introductorio de ajuste de datos mediante red neuronal simple
% x: entradas; t: salidas deseadas
clc; clear all ; close all
[x1,t1] = simplefit_dataset;
% x1 = simplefitInputs; t1 = simplefitTargets;
% Seleccionamos algoritmo de entrenamiento, para ayudas tipear help nntrain
% opción 'trainlm': Levenberg-Marquardt, opción 'trainbr' algoritmo de regulación Bayesiana.
% 'trainscg': propagación hacia atrás, utiliza gradiente conjugado.
trainFcn = 'trainlm'; % Levenberg-Marquardt backpropagation.
% Construimos la red de ajuste de datos
neuronas_capa_oculta = 10;
net1 = fitnet(neuronas_capa_oculta,trainFcn);
% Seleccionamos funciones de pre y post procesamiento para las entradas y salidas
% Tipear help nnprocess para ver ayudas
net1.input.processFcns = {'removeconstantrows','mapminmax'};
net1.output.processFcns = {'removeconstantrows','mapminmax'};
% Configuración de los datos de entrenamiento, validación y testeo
% Para obtener ayudas de funciones tipear: help nndivision
net1.divideFcn = 'dividerand'; % Divide los datos en forma aleatoria
net1.divideMode = 'sample'; % Divide cada muestra
net1.divideParam.trainRatio = 70/100; net1.divideParam.valRatio = 15/100;
net1.divideParam.testRatio = 15/100;
% Seleccionamos función de Performance
% Para ayudas tipear: help nnperformance
net1.performFcn = 'mse' ; % Error cuadrático medio MSE
% Elegimos todas las funciones a graficar
% Para ayudas tipear: help nnplot
net1.plotFcns = { 'plotperform', 'plottrainstate', 'plotregression', 'ploterrhist', 'plotfit' } ;
% Entrenamos la red
[net1,tr] = train(net1,x1,t1);
% Testeamos la red
y1 = net1(x1);
e = gsubtract(t1,y1);
Performance1 = perform(net1,t1,y1)
% Recalculamos Performance de Entrenamiento, Validación y testeo
trainTargets = t1 .* tr.trainMask{ 1}; valTargets = t1 .* tr.valMask{ 1}; %
testTargets = t1 .* tr.testMask{1}; trainPerformance1 = perform(net1, trainTargets,y1) %
valPerformance1 = perform(net1, valTargets,y1)
testPerformance1 = perform(net1, testTargets,y1)
% Mostramos la red neuronal y Graficamos
view(net1)
% Agregar comentarios en función de los resultados que se quieran graficar
figure; plotperform(tr); figure, plottrainstate(tr)
figure, ploterrhist(e) ; figure, plotregression(t1,y1)
figure, plotfit(net1,x1,t1)
% Agregar comentarios según las funciones que se quieran generar
genFunction(net1,'NeuralNetworkFunction_1');
y = NeuralNetworkFunction_1(x1);
% Generamos otra función de red neuronal, solo soporta matrices de entrada
% (no acepta array de celdas)
genFunction(net1,'NeuralNetworkFunction_2','MatrixOnly','yes');
y = NeuralNetworkFunction_2(x1);
% Generamos diagrama Simulink
gensim(net1);

```

En las últimas líneas del programa se genera el diagrama Simulink de la red neuronal, se puede expandir cada bloque, ver Fig. 53.

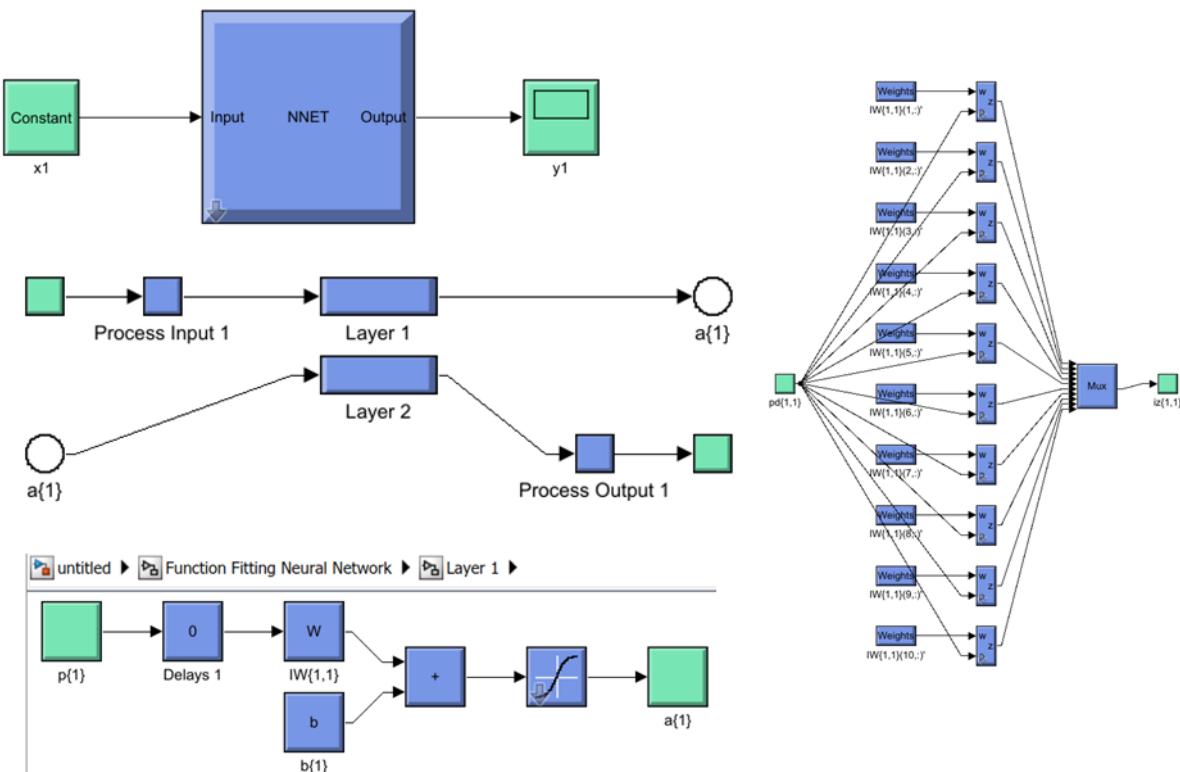


Fig. 53 – Ejemplo de Red Neuronal con diagrama Simulink

Ejercicio 4.3

Ajuste de datos con redes neuronales estáticas mediante script en Matlab

La herramienta nnstart solo se recomienda para una etapa inicial ya que puede generar redes con alta carga computacional y resultados que no sean óptimos o que no sean totalmente coherentes. Conviene trabajar con script y analizar cada línea del programa.

El ejercicio anterior se resuelve con el siguiente código:

```
close all; clear all ; clc
[x1,t1] = simplefit_dataset;
net1 = feedforwardnet;
%net = feedforwardnet(15);
net1 = configure(net1,x1,t1);
net1 = init(net1);
view(net1)
net1.trainFcn = 'trainlm';
[net1,tr1] = train(net1,x1,t1);
plotperf(tr1)
```

Ejercicio 4.4

Reconocimiento de Patrones: Clasificación de datos con redes neuronales

En este ejemplo se clasifican datos, donde las salidas son clases (grupos). Se utiliza la herramienta nprtool de Matlab. Esta herramienta se debe utilizar con mucho cuidado y analizar cada línea del código generado.

Se pide:

- Abrir la herramienta de Matlab® de Clasificación (comando nprtool), seleccionar los siguientes datos “simpleclass_dataset”. Se pueden cargar datos de ejemplo de Matlab® o utilizar datos propios. Luego seleccionar porcentajes de datos para entrenamiento, validación y testeо (por ejemplo 70-15-15 respectivamente). Asignar 10 neuronas para la capa oculta. Pulsar el botón para “entrenar” la red. Presionar los comandos “Plot confusion” y “Performance” (en nntraintool). En la Fig. 54 se muestran los resultados de la clasificación de las 4 categorías, todos los datos se clasifican correctamente (100% correcto).
- En la pantalla final de generar el código de programa (script) correspondiente mediante el comando “Advanced Script”.
- Reducir la cantidad de neuronas de la red para analizar el cambio de Performance.

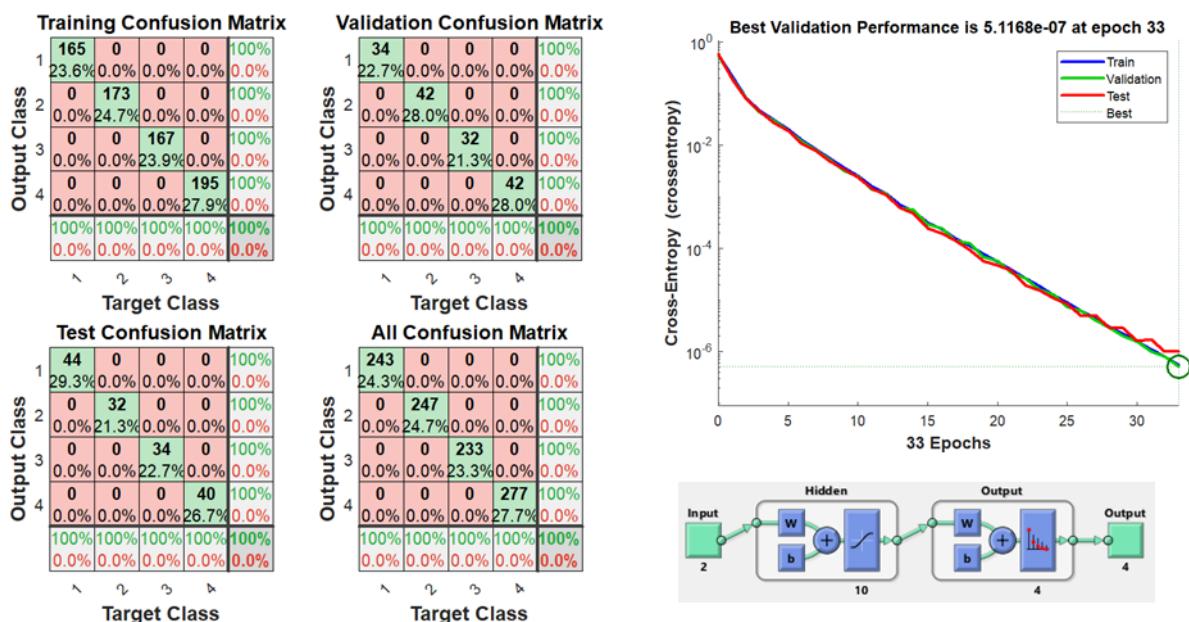


Fig. 54 – Resultados de la Clasificación de los datos en 4 categorías. Izquierda) Matrices de Confusión, Derecha) Topología y Performance de la red

A continuación, se muestra el programa modificado para una mejor comprensión, a su vez se modifican las primeras líneas para importar los datos correspondientes. Se dispone de un script independiente de la herramienta nprtool.

```
% Ejemplo de Clasificación - Reconocimiento de Patrones con redes neuronales
% Importamos los datos; x: entradas, t: categoría deseada
clear all; clc; close all;
[ x1, t1 ] = simpleclass_dataset;
% Elegimos función de entrenamiento (tipar: help nntrain)
% 'trainlm': Levenberg-Marquardt, 'trainbr': regulación Bayesiana.
% 'trainscg': Algoritmo de propagación hacia atrás mediante gradiente conjugado.
func_train = 'trainscg'; % Gradiente conjugado backpropagation.
% Creamos la red neuronal de Clasificación
neuronas_capa_oculta = 10;
net1 = patternnet(neuronas_capa_oculta, func_train);
% Seleccionamos funciones de pre y de post procesamiento para las entradas y salidas
% Para mostrar la lista de todas las funciones tipar: help nnprocess
net1.input.processFcns = { 'removeconstantrows' , 'mapminmax' } ;
% Configuración de los datos de entrenamiento, validación y testeo
% Para obtener ayudas de funciones tipar: help nndivision
net1.divideFcn = 'dividerand'; % Divide datos en forma aleatoria
net1.divideMode = 'sample';
net1.divideParam.trainRatio = 70/100 ; net1.divideParam.valRatio = 15/100 ;
net1.divideParam.testRatio = 15/100 ;
% Seleccionamos la función de Performance.
% Para obtener ayudas tipar: help nnperformance
net1.performFcn = 'crossentropy'; % Entropía cruzada
% Elegimos todas las funciones que queremos graficar
% Para ayudas tipar: help nnplot
net1.plotFcns = {'plotperform', 'plotroc', 'plottrainstate', ...
    'plotconfusion', 'ploterrhist'};
% Entrenamos la red
[net1,tr1] = train(net1,x1,t1);
% Testeamos la red
yy = net1(x1);
e = gsubtract(t1,yy);
performance = perform(net1,t1,yy)
t_ind = vec2ind(t1);
y_ind = vec2ind(yy);
percentErrors = sum(t_ind ~= y_ind)/numel(t_ind);
% Volvemos a calcular Performance de Entrenamiento, Validación y testeo
trainTargets = t1 .* tr1.trainMask{1};
valTargets = t1 .* tr1.valMask{1};
testTargets = t1 .* tr1.testMask{1};
trainPerformance1 = perform(net1,trainTargets, yy)
valPerformance1 = perform(net1,valTargets,yy)
testPerformance1 = perform(net1,testTargets,yy)
% Mostramos la red
view(net1)
% Graficamos, agregar comentarios si no quiere mostrar todos los resultados
figure; plotperform(tr1); figure, plottrainstate(tr1)
figure, ploterrhist(e) ; figure, plotconfusion(t1,yy)
figure, plotroc(t1,yy)
```

```
% Desarrollo, generamos función de red neuronal
genFunction(net1,'NeuralNetworkFunction_1');
y = NeuralNetworkFunction_1(x1);
% Generamos función de red neuronal para matrices de entrada (no soporta array de celdas)
genFunction(net1,'NeuralNetworkFunction_2','MatrixOnly','yes');
y = NeuralNetworkFunction_2(x1);
% Generamos diagrama Simulink
gensim(net1);
```

Ejercicio de Agrupamiento de datos con redes neuronales:

En la sección de redes neuronales con mapas autoorganizados (SOM) se disponen de ejemplos de agrupamiento de datos.

Reconocimiento estadístico de patrones

El Reconocimiento estadístico de patrones (REP) está basado en la teoría de probabilidad y estadística. En esta técnica se supone que el conjunto de mediciones tiene distribuciones de probabilidad conocidas. A partir de estas distribuciones se realiza el reconocimiento (Papoulis, 2002), (Röck, 2008), (Scott, 2006), (Webb, 2011).

El procesamiento de señales comprende las etapas de adquisición de datos, extracción de parámetros de las señales y clasificación de las muestras. Se pueden utilizar distintas técnicas de reconocimiento de patrones para evaluar los resultados. El reconocimiento de patrones involucra todas las fases de investigación y formulación del problema, también involucra el análisis de datos a través de la discriminación y clasificación para analizar y evaluar los resultados (Theodoridis, 2010).

En la Fig. 55 se representa un esquema de reconocimiento de patrones en un sistema de medición, que se puede separar en distintas etapas:

- i- Sistema de adquisición de datos
- ii- Sistema de extracción de parámetros
- iii- Clasificadores
- iv- Estrategia en la toma de decisiones

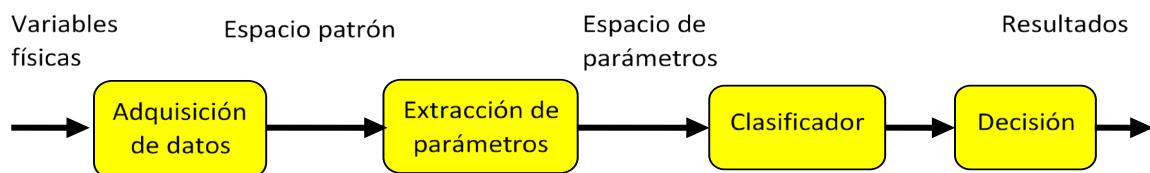


Fig. 55 – Esquema general para el reconocimiento de patrones

Para el caso de mediciones que están separadas en distintos grupos, las salidas de los clasificadores no siempre son digitales indicando el grupo correspondiente de la medición realizada,

a veces solo indican un valor analógico. En estos casos resulta necesario establecer límites y métodos para determinar la salida adecuada. En la Fig. 56 se muestra un esquema básico de reconocimiento de patrones, donde las distintas etapas pueden tener realimentación de la salida (Su, 2021).

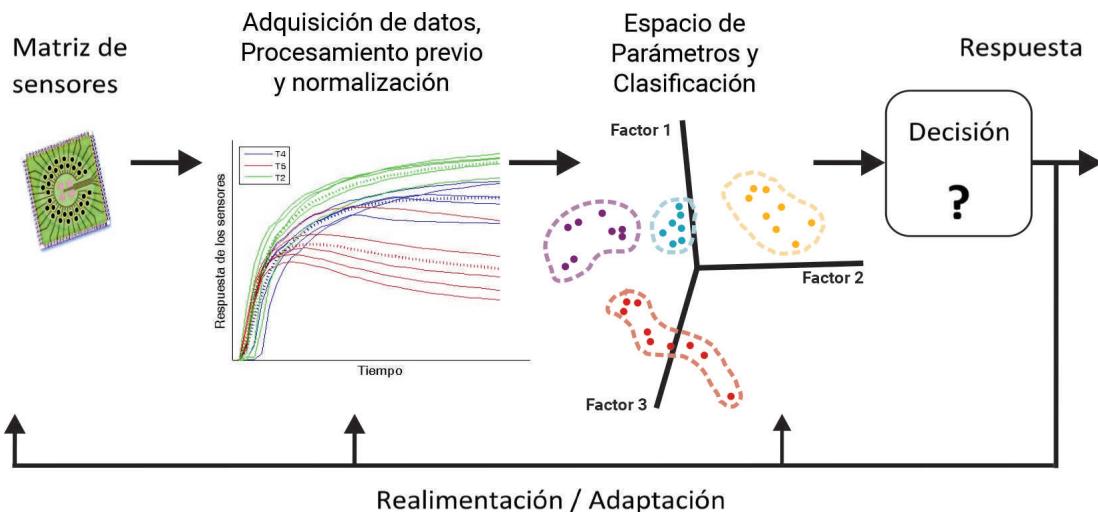


Fig. 56 – Esquema general de reconocimiento de patrones con realimentación

Existe una variedad enorme de técnicas de reconocimiento de patrones. A continuación, solo se describen brevemente algunas de ellas.

Técnicas de reconocimiento de patrones

Existen distintos métodos que se pueden separar en 2 grupos:

- Reconocimiento Estadísticos de Patrones (REP)
- Métodos con inteligencia artificial (AI) que incluye uso de redes neuronales y lógica difusa.

Los datos se pueden separar en variables parcialmente independientes (variables de medición) y en variables dependientes (clases o grupos).

A su vez, existen algoritmos de reconocimiento de patrones supervisados y no supervisados. Los métodos de exploración de datos se consideran no supervisadas, mientras que los métodos de clasificación donde se entrena con salidas conocidas son supervisados (Pedregosa, 2011).

- Los algoritmos supervisados asignan un descriptor o salida al vector de datos de entrada correspondiente a las mediciones. Este descriptor (salida o respuesta del sistema) es un vector que se asigna en el entrenamiento. Teniendo un conjunto de mediciones separadas en grupos o clases según su semejanza, el vector descriptor indica el grupo de pertenencia de cada medición. Los algoritmos se tienen que “entrenar” y establecer sus parámetros internos. Finalizado el aprendizaje o entrenamiento, se ingresan datos de una medición con grupo desconocido y el algoritmo clasifica los datos en base al entrenamiento realizado.
- En los algoritmos no supervisados, los descriptores no son asignados previamente, los algoritmos se entrena y ajustan sus parámetros en base a algún tipo de similitud encontrada.

A continuación, se nombran algunos métodos REP (Subasi, 2020)

Análisis Multivariado de Datos

- Métodos de reducción de dimensión y proyección de datos: PCR, PCA, LDA

Métodos de agrupamiento

- Agrupamiento jerárquico y K-means

Clasificadores

- Clasificador Bayesiano óptimo
- Análisis discriminante lineal y cuadrático
- Regresión logística
- Máquinas de soporte vectorial

Los métodos estadísticos se consideran paramétricos, ya que se supone que los datos pueden describirse con funciones de densidad de probabilidad. En esta categoría de métodos tenemos análisis de componentes principales (PCA), análisis de factores discriminantes (DFA), análisis de función de densidad de probabilidad mediante teorema de Bayes, método de regresión de cuadrados mínimos parciales (PLS) y algoritmos de separación de grupos como clúster jerárquico y k-means (Yu, 2008).

Las técnicas de inteligencia artificial (AI), incluye técnicas con enfoques intuitivos que están inspirados en modelos biológicos (Proakis, 1998), (Pouliarikas, 2019). Se pueden clasificar en tres subgrupos:

- Redes neuronales artificiales (en inglés artificial neural network: ANN), incluyen propagación hacia atrás de errores (back propagation: BP), perceptrón de múltiples capas (MLP) y redes con funciones de base radial (conocidas como radial basis function network: RBF), mapas autoorganizados (Self Organizing Maps: SOM), Learning vector quantization (LVQ), redes dinámicas recurrentes y sistemas adaptativos (Diniz, 2020).
- Algoritmos de lógica difusa (conocidos como Fuzzy logic) y normas o razonamiento difusos
- Algoritmos genéticos (GA) usados para selección de parámetros.

En la Fig. 57 se muestra un ejemplo de análisis de datos donde se separan dos grupos diferentes (dos clases distintas). Se procesan los datos con software Matlab®. A través del método PCA, se proyectan los datos en dos dimensiones. Luego se calculan tres clasificadores distintos: a) lineal, b) cuadrático y c) red neuronal de propagación hacia atrás. Posteriormente se elige el mejor clasificador (Beale, 2020).

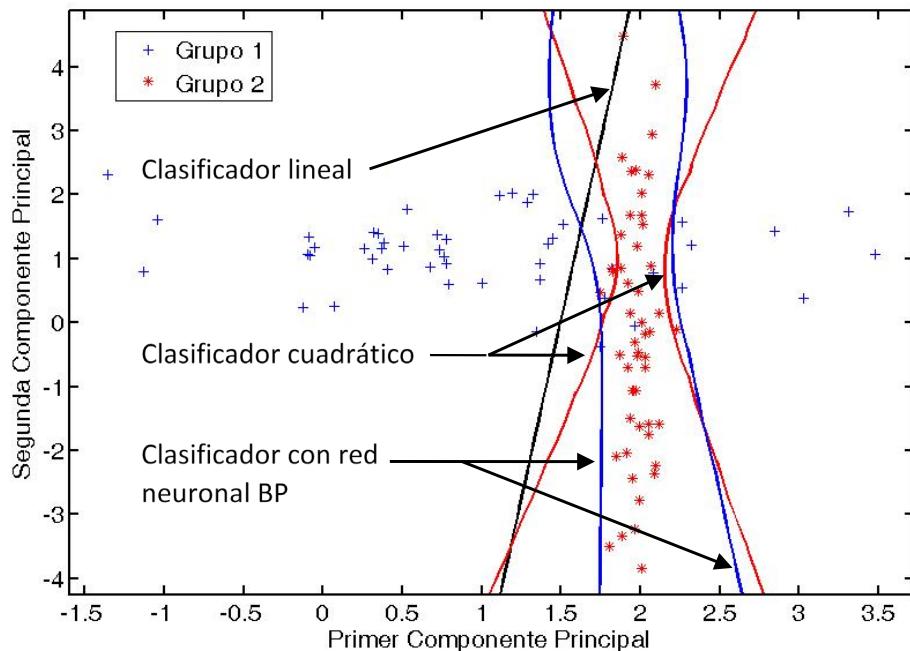


Fig. 57 – Análisis de datos combinando técnicas de PCA con diferentes clasificadores

Similitud

La similitud es fundamental para la definición de clúster (grupo o conjunto) de objetos. En la similitud se miden las distancias que se tienen entre distintos objetos (mediciones o grupo de mediciones) para encontrar relaciones. Se selecciona una función de distancia definida para encontrar determinados patrones (Del Brío, 2007). Existe una gran variedad de parámetros y de escalas. La elección de la función de distancia se debe elegir cuidadosamente. La métrica más utilizada para medir patrones continuos es la distancia Euclidiana, ver ecuaciones (4.1) y (4.2).

$$d(X_i, X_j) = \left(\sum_{k=1}^d (X_{ik} - X_{jk})^2 \right)^{\frac{1}{2}} \quad (4.1)$$

$$d(X_i, X_j) = \|X_i - X_j\| \quad (4.2)$$

La distancia Euclidiana se utiliza mucho para evaluar la cercanía entre objetos de dos y tres dimensiones espaciales. Presenta el inconveniente de la tendencia de las variables de mayor escala a dominar a las demás. Este problema se soluciona mediante una correcta normalización o mediante métodos de ponderación.

La métrica de correlación lineal también puede distorsionar las mediciones de las distancias. Estas distorsiones muchas veces se pueden reducir mediante el uso de la distancia de Mahalanobis elevada al cuadrado, ver ecuación (4.3).

$$d_M(X_i, X_j) = (X_i - X_j) \Sigma^{-1} (X_i - X_j)^T \quad (4.3)$$

Donde X_j y X_i son vectores fila, $(X_i - X_j)^T$ es la transpuesta del vector $(X_i - X_j)$. Siendo Σ^{-1} la inversa de la matriz de covarianza de las muestras; mediante d_M se asignan diferentes pesos a los parámetros basado en sus varianzas y sus correlaciones.

Clúster de particiones

Un clúster, o grupo, es un subconjunto del conjunto de datos completos, donde los elementos que lo componen deben tener cierta similitud entre sí. Los algoritmos de clúster de particiones intentan segmentar los datos y crear estructuras de dendrogramas (similares entre sí). Un dendrograma es un diagrama o una representación gráfica con datos en forma de árbol (Dendro: árbol). Este diagrama organiza los datos en distintas subcategorías que se van separando entre sí. Surge el problema que no está especificado el número de clúster, se debe asignar antes de empezar los cálculos. Las técnicas de fraccionamiento para generar clúster generalmente están basadas en la minimización de las distancias.

Los algoritmos de clúster son no supervisados. Existen algoritmos de clúster jerárquico y clúster k-means.

Clúster jerárquico

El análisis por clúster jerárquico (HCA) intenta separar los datos en grupos, basado en mediciones semejantes. Inicialmente cada dato representa su propio grupo, luego se empieza a juntar objetos, reduciendo el umbral de decisión para poder declarar dos o más objetos miembros de un mismo clúster (grupo). Si seguimos juntando en grupos, cada vez más objetos están vinculados entre sí, donde se forman grupos más grandes con elementos u objetos más distintos entre sí.

Se utilizan dendrogramas para representar el resultado de los clústers jerárquicos, donde se distinguen los distintos grupos. Los grupos se pueden vincular o asociar de muchas maneras distintas. Las tres formas más utilizadas son:

- 1) Mediante vinculación simple (K-NN o vecinos más cercanos: “K nearest neighbors” en inglés), donde la distancia entre dos grupos está determinada por la distancia entre los dos objetos más cercanos pertenecientes a distintos grupos.
- 2) Vinculación completa: se toma como distancia entre dos grupos la distancia correspondiente a los dos objetos más lejanos de cada grupo.
- 3) Promedio de grupo: se toma como distancia entre dos grupos distintos la distancia promedio de todos los objetos pertenecientes a distintos grupos.

En la Fig. 58 se muestra un dendrograma con distintas clases. Cada medición tiene asignado un número de muestra, en el eje de coordenadas se muestran dichos números. El eje de abscisas representa las distancias entre los distintos grupos. Luego de graficar las distancias, se traza una recta horizontal, en este ejemplo se distinguen tres grupos distintos. Si se sube o baja la posición de la recta, cambiará la cantidad de grupos (clúster).

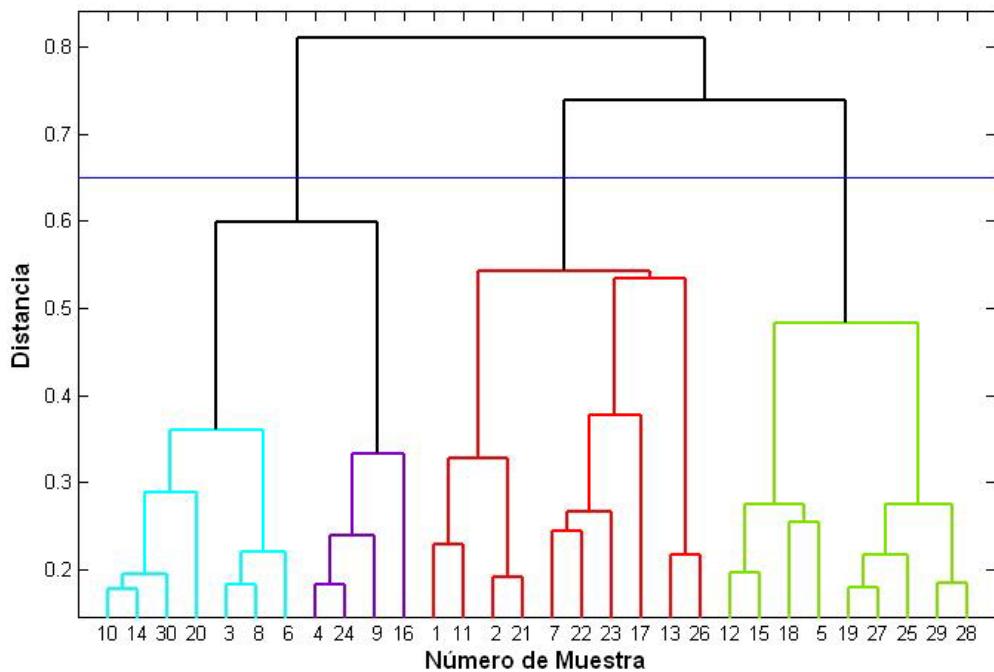


Fig. 58 – Ejemplo de dendrograma donde se traza línea horizontal para distinguir tres grupos.

Análisis de Componentes Principales (PCA)

El análisis de componentes principales (PCA) reduce la dimensión de los datos, es un método de proyección lineal no supervisado. Este método extrae parámetros, toma como entrada al vector de mediciones $X \in R^n$ y lo convierte en un nuevo espacio de descripción de parámetros $Z \in R^m$ de manera de facilitar su visualización. Al ser $m < n$ reducimos la dimensión de los datos. Para graficar en 2 o 3 dimensiones, generalmente m tiene valor 2 o 3, obteniendo un nuevo espacio reducido.

La entrada al algoritmo PCA son los datos de entrenamiento $T_X = \{X_1, X_2, \dots, X_l\}$ en forma de vectores, tiene dimensión n correspondiente al espacio de entrada R^n , siendo l la cantidad de muestras. Se obtiene un conjunto de vectores $T_Z = \{Z_1, Z_2, \dots, Z_l\}$ que es una representación de T_X de menor dimensión, correspondiente al espacio R^m con dimensión m . El conjunto de vectores T_Z se calcula con la proyección lineal ortonormal, ver ecuación (4.4).

$$Z = W^T X + b \quad (4.4)$$

Donde la matriz $W[n \times m]$ y el vector de desviación $b \in R^m$ son parámetros de proyección.

Operando e invirtiendo la última ecuación obtenemos el vector reconstruido: $T_{\tilde{X}} = \{\tilde{X}_1, \tilde{X}_2, \dots, \tilde{X}_l\}$ que se calcula mediante la proyección inversa (4.5).

$$\tilde{X} = W(Z - b) \quad (4.5)$$

Para calcular el error cuadrático medio correspondiente a la reconstrucción utilizamos la ecuación (4.6).

$$\varepsilon_{MS}(W, b) = \frac{1}{l} \sum_{i=1}^l \|X_i - \tilde{X}_i\|^2 \quad (4.6)$$

PCA realiza una proyección ortonormal lineal de manera de minimizar el error $\varepsilon_{MS}(W, b)$

Los parámetros (W, b) corresponden a la solución de error mínimo, ecuación (4.7).

$$(W, b) = \underset{W', b'}{\operatorname{argmin}} \varepsilon_{MS}(W', b') \quad (4.7)$$

Con $\langle w_i, w_j \rangle = \delta_{(i,j)}, \forall i, j$

Donde $w_i, i = 1, \dots, m$ son vectores columna de la matriz $W = [w_1, w_2, \dots, w_m]$ y $\delta_{(i,j)}$ es la función conocida como delta de Kronecker.

Planteando la solución de mínimo error se obtiene $W = [w_1, w_2, \dots, w_m]$ que contiene m autovectores correspondientes a la matriz de covarianza de la muestra, con los autovalores de mayor valor. El vector b es igual a $W^T \mu$ donde μ es el valor medio de los datos de analizados de entrenamiento.

Los datos se suelen normalizar para obtener una mejor discriminación. En la Fig. 59 se muestran los efectos de la normalización por autoescala y del centrado en análisis PCA.

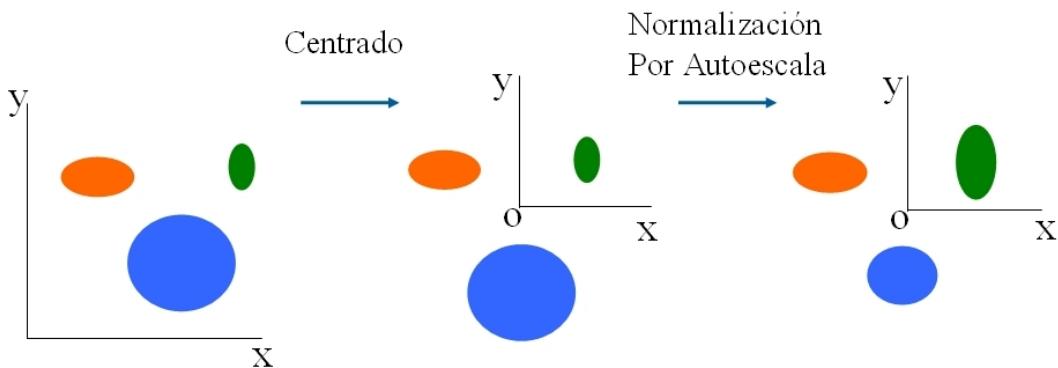


Fig. 59 – Efectos del centrado y de la normalización por autoescala para el análisis PCA

En la Fig. 60 se muestra un gráfico de PCA procesado con software Matlab®. Se genera en forma aleatoria un conjunto de datos de 2 dimensiones, con distribución Gaussiana. Estos datos se encuentran separados en 3 grupos (Grupo 1, 2 y 3) y se grafican en 2 dimensiones (llamados eje X y eje Y o PC1 y PC2). Se aplica PCA para obtener un subespacio nuevo de menor dimensión, 1 solo eje. Se aproximan los datos de entrada a este subespacio, minimizando el error de reconstrucción. Luego se grafican los datos reconstruidos en el subespacio de dimensión 1 con su primer componente principal. De esta manera se toma un espacio de dimensión 2 y se lo reduce a dimensión 1.

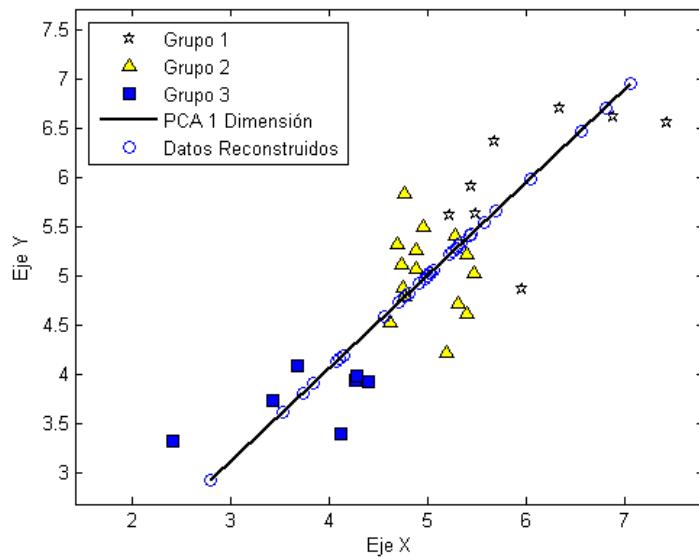


Fig. 60 – Ejemplo de PCA. Datos originales en 2 dimensiones (eje x y eje y) y datos reconstruidos en 1 dimensión (recta).

Es decir que se pueden tomar datos de 2 variables, cada variable corresponde a los ejes del gráfico de 2 dimensiones, luego se aplica PCA y se grafica en 1 o en 2 dimensiones (Demuth, 2018).

En la Fig. 61 mostramos otro ejemplo de PCA para clasificar distintos aromas de té verde. Los datos de entrada corresponden a un espacio de dimensión 32, se reduce a dimensión 2 y se grafica. Los porcentajes de varianza se indican en el gráfico en cada eje (eje PC1 y PC2 componente principal 1 y 2, respectivamente). En este ejemplo tenemos PC1: 56,16 % y PC2: 34,27 %.

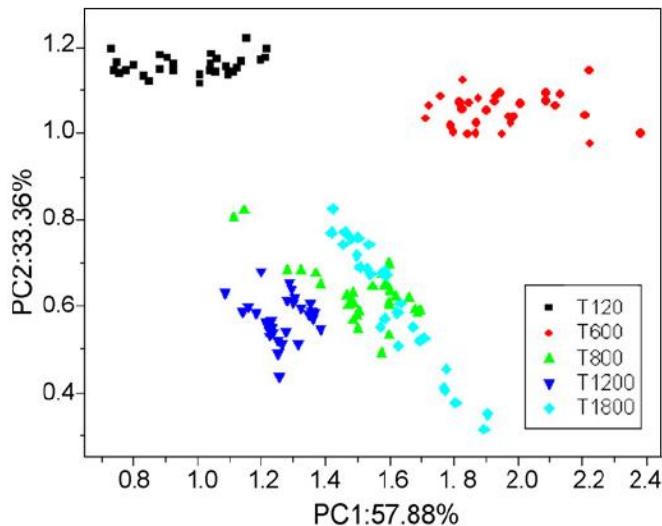


Fig. 61 – Ejemplo de PCA utilizado para clasificación de té.

Clasificadores

Una vez realizada la extracción de parámetros o reducción de dimensión, que se realiza con técnicas de PCA u otros métodos, se clasifican las muestras. Cabe aclarar que el proceso de reducción puede no realizarse y se puede clasificar directamente las muestras.

El límite entre un extractor de parámetros y el clasificador resulta casi arbitrario. El extractor de parámetros ideal debe crear un conjunto de datos significativos que reduzcan trivialmente el trabajo y la carga de los clasificadores. A su vez, un clasificador universal no necesita utilizar un extractor de parámetros sofisticado. El clasificador toma el vector de parámetros proveniente del extractor de parámetros, y asigna determinadas categorías. La dificultad del clasificador depende de la variación de los parámetros que tengan los objetos de la misma categoría. Todo esto se considera en forma relativa respecto a las diferencias que existan entre los parámetros de los objetos de otras categorías.

Se puede medir la eficiencia de los clasificadores mediante la tasa de error de clasificación, que se calcula como el porcentaje de muestras que se asignan a categorías incorrectas. El último paso de un sistema de clasificación generalmente es tomar una decisión, se debe asignar una clase (o grupo) al patrón de entrada.

Los clasificadores se utilizan mucho en determinados ambientes industriales. Se proponen los siguientes requerimientos que debe tener un clasificador ideal:

- Alta exactitud: debe tener la menor cantidad posible de clasificaciones erróneas o incorrectas.
- Rápido para el análisis en tiempo real. El algoritmo del clasificador debe tener la mínima demora posible.
- Entrenamiento simple. En muchas aplicaciones se deben entrenar periódicamente los algoritmos. Este procedimiento tiene que ser rápido y simple.
- Bajos requerimientos de hardware, de memoria y de baja carga computacional, para que se puedan usar en dispositivos móviles.
- Robustos a las mediciones falsas o anomalías. Debe reducir los errores de clasificación debido a las posibles mediciones erróneas o con gran dispersión.
- Generar un índice de probabilidad del resultado o incertezza de las mediciones realizadas. Muchas aplicaciones requieren este índice.

Medición y validación cruzada

La validación cruzada se utiliza para evaluar los resultados de un procesamiento o análisis estadístico, de manera de garantizar que los resultados son independientes de la partición que se tome entre los datos de entrenamiento y los datos de prueba. Se partitionan varias veces los datos de las entradas en conjuntos de entrenamiento y conjuntos de prueba. Para cada partición se evalúan los clasificadores utilizados. Luego se puede calcular el valor medio de las medidas de evaluación para diferentes clasificadores.

Se utiliza mucho en aplicaciones donde el objetivo principal es predecir los resultados, se necesita estimar cuan preciso es el modelo final que se utilizará en la práctica. Es una técnica muy utilizada, sirve para validar y comparar modelos.

Mediante validación cruzada se reducen los errores en la evaluación de modelos, generados por tomar conjuntos pequeños de datos de entrenamiento y de prueba.

La rutina de validación cruzada toma el conjunto de datos y omite muestras, una o más por vez. Entrena al algoritmo de clasificación usando los datos restantes. Posteriormente clasifica las observaciones que fueron omitidas. Luego de varias repeticiones se obtiene el valor medio del índice correspondiente a las clasificaciones correctas (Pytorch, 2021).

Análisis de funciones discriminantes (DFA)

Las técnicas de DFA son supervisadas, es decir que antes de clasificar las muestras desconocidas se entran los algoritmos con un conjunto de datos o muestras conocidas. Existen dos tipos de DFA: cuadráticos y lineales. En ambos casos se supone que los datos tienen una distribución normal.

El análisis discriminante lineal (LDA), calcula una función lineal discriminante (LDF), que resulta de una combinación lineal de todas las variables originales $X = (x_1, x_2, \dots, x_p)$. El centro de las clases se estima mediante la ecuación (4.8).

$$\mu_k = \frac{1}{N_K} \sum_{i=1}^{N_K} x_i \quad (4.8)$$

Donde N_k corresponde al número de muestras de cada clase; con $k = 1, 2, \dots, K$. Siendo K la cantidad de clases.

El valor real de la distribución π es desconocido y se estima como $\pi_k = \frac{N_k}{N}$, donde N es el número total de muestras. Se parte de la suposición que la matriz de covarianza Σ resulta igual para todos los grupos. Por lo que se toma una sola matriz común a todos los grupos, la cual se estima con la siguiente ecuación (4.9):

$$\Sigma = \frac{1}{N - K} \sum_{k=1}^K \left(\sum_{i=1}^{N_K} (x_i - \mu_k) \cdot (x_i - \mu_k)^T \right) \quad (4.9)$$

La función lineal discriminante es $\delta_k(X)$, ecuación (4.10).

$$\delta_k(X) = X^T \cdot \sum_{k=1}^{-1} \mu_k - \frac{1}{2} \mu_k^T \cdot \sum_{k=1}^{-1} \mu_k + \log(\pi_k) \quad (4.10)$$

Siendo Σ^{-1} la inversa de la matriz de covarianza, X^T es la transpuesta de la matriz X y μ_k^T es la transpuesta del vector μ_k . El límite de separación entre clases está representado por ecuaciones en función de X . El límite entre dos clases, por ejemplo, clases A y B, se puede escribir con la siguiente ecuación (4.11):

$$\log\left(\frac{\pi_A}{\pi_B}\right) - \frac{1}{2} (\mu_A + \mu_B)^T \cdot \sum_{k=1}^{-1} (\mu_A - \mu_B) + X^T \cdot \sum_{k=1}^{-1} (\mu_A - \mu_B) = 0 \quad (4.11)$$

Donde μ_A y μ_B son los centros de los vectores A y B , respectivamente. Entonces se tienen $K - 1$ funciones discriminantes disponibles, que se pueden graficar para visualizar la separación de los datos.

Análisis discriminantes lineales mediante método de Fisher

En el método de Fischer para separar dos clases: C_1 y C_2 se busca una combinación lineal de las variables que maximice el factor J_F que relaciona las varianzas entre distintas clases y las varianzas internas de las clases.

Esta relación J_F se muestra en la ecuación (4.12), donde se analizan los valores de w para obtener máximo valor de J_F .

$$J_F = \frac{|w^T(m_1 - m_2)|^2}{w^T S_w \cdot w} \quad (4.12)$$

Siendo m_1 y m_2 son los valores medios de las clases y S_w representa la suma de las matrices de covarianza interna de las clases dadas por la ecuación (4.13).

$$S_w = \frac{1}{n-2} \cdot (n_1 \cdot \Sigma_1 + n_2 \cdot \Sigma_2) \quad (4.13)$$

Donde Σ_1 y Σ_2 son las matrices de covarianza interna de las clases C_1 y C_2 , respectivamente. La cantidad de muestras de cada clase se indica con n_1 y n_2 . Para dos clases, $n = n_1 + n_2$ es la cantidad total de muestras.

Para calcular el valor de w que maximice J_F se calcula la derivada de J_F respecto de w y se iguala la derivada a cero. Despejando el valor de w se obtiene la ecuación (4.14).

$$w = S_w^{-1} \cdot (m_1 - m_2) \quad (4.14)$$

Mediante la ecuación (4.15) se plantea un límite que separe las 2 clases.

$$w^T \cdot x + w_0 \geq 0 \quad (4.15)$$

La variable w_0 modifica el límite separación, se puede calcular mediante la ecuación (4.16).

$$w_0 = -\frac{1}{2} \cdot (m_1 - m_2)^T \cdot S_w^{-1} \cdot (m_1 - m_2) - \log \left[\frac{p(c_1)}{p(c_2)} \right] \quad (4.16)$$

Siendo $p(c_1)$ la prioridad de la clase C_1 y $p(c_2)$ la prioridad de la clase C_2 . Donde se supone distribución normal en todo el conjunto de datos.

Si $w^T \cdot X + w_0 \geq 0$ entonces una medición llamada X se le asigna a la clase C_1 , caso contrario se asigna la clase C_2 .

Método K-NN

El nombre K-Nearest-Neighbors (K-NN), en español quiere decir “vecino más cercano”. Es un método de clasificación del tipo supervisado basado en un conjunto de datos de entrenamiento conocido. Estima la función de densidad de probabilidad (FDP) o directamente la probabilidad de que un elemento o muestra X pertenezca a una determinada clase llamada C, utiliza información proveniente del conjunto de datos de entrenamiento. Los pasos de clasificación son los siguientes:

- Para el dato o elemento X se calculan las distancias entre este elemento X y sus vecinos más cercanos. Las distancias se deben calcular eligiendo la métrica más adecuada. Se toma una cantidad K de vectores (distancias) de menor tamaño.
- El espacio completo es particionado en diferentes regiones por localizaciones y etiquetas según los datos de entrenamiento. Entonces al elemento X se le asigna la clase C obtenida de una votación según las clases a las que pertenezcan sus k vecinos más cercanos. Esto quiere decir que al elemento X, se le asigna la clase C, siendo C la clase de mayor frecuencia entre los K elementos de entrenamiento más próximos, los más cercanos.

La distancia entre dos elementos o mediciones denominados x e y , se puede representar mediante el vector $d(x, y)$. A modo de ejemplo, la métrica Euclidiana calcula la distancia según la ecuación (4.17). Esta métrica funciona bien si se tienen escalas similares en todas las componentes del vector de medición. Si no fuera así, se necesita normalizar la medición.

$$d(x, y) = |x - y| \quad (4.17)$$

Para realizar una clasificación con K-NN, primero se debe ingresar la cantidad de vecinos más cercanos es decir K, la métrica para calcular la distancia y el conjunto de entrenamiento. Posteriormente se ingresa la medición X desconocida o incógnita y se determina la clase C a la cual pertenece.

En este método se supone que las muestras o vecinos más cercanos nos brindan la mejor clasificación y se utilizan todas las variables de la medición. El problema de esta suposición es que posiblemente muchas variables sean irrelevantes y perjudiquen la clasificación. Por ejemplo, dos variables relevantes podrían perder peso o significancia respecto a muchas otras variables irrelevantes. Para corregir este problema se pueden asignar distintos pesos para cada una de las distancias a las variables. Dando mayor importancia a las variables más relevantes. Otra forma consiste en determinar o ajustar los pesos utilizando ejemplos conocidos de entrenamiento. Antes de asignar los pesos es recomendable eliminar las variables que puedan perjudicar los resultados y se consideran irrelevantes.

Existen distintos criterios para la elección de K , depende fundamentalmente de los datos. En principio, valores grandes de K reducen el ruido en la clasificación, pero se mezclan las clases similares. Un valor adecuado de K se puede seleccionar mediante una optimización de uso para determinados datos. Se puede usar $K = 1$, en este caso especial la clase es predicha es la clase más cercana a la muestra, se denomina Algoritmo del vecino más cercano (en inglés: Nearest Neighbors Algorithm). Si se tiene ruido o variables irrelevantes, este algoritmo puede funcionar mal. O puede fallar si las escalas no son consistentes con respecto a los resultados que se quieren obtener.

Ejercicio 4.5

Método K-NN

Analizar el siguiente código de clasificación de datos con Método K-NN

```
% Entrenamos el clasificador k-NN: k-Nearest Neighbors Classifier con los datos Fisher Iris de
% Matlab®. Utilizamos k=5, siendo k el número de vecinos cercanos en el predictor
load fisheriris % cargamos los datos
XX = meas;
YY = species;
% XX es una matriz que contiene 4 mediciones de pétalos de 150 plantas Iris
% YY contiene las especies correspondientes a cada medición
% Entrenamos k-NN. Normalizamos con predictor no categórico, obtenemos los
% resultados en la estructura Class. Observar resultados en ventana de
% comandos de Matlab®
Class = fitcknn(XX,YY,'NumNeighbors',5,'Standardize',1)
% Accedemos a algunas propiedades
Class.ClassNames
Class.Prior
YY(5) % Mostramos la etiqueta original de la muestra 5
Class.Y(5) % Mostramos la etiqueta predecida de la muestra 5
% Predecimos la muestra 5 modificada
label1 = predict(Class,XX(5,:)*1.1) % modificamos levemente
label2 = predict(Class,XX(5,:)*5) % modificamos ampliamente
```

Analizando las últimas 2 líneas, label1 tiene como resultado {'setosa'}. Y label2 tiene como resultado {'virginica'}

Análisis de datos y sensores

El análisis de datos y aplicaciones no solo se limita en implementar un clasificador, sino que comienza en la adquisición de datos. Se deben seleccionar los sensores adecuados, preprocesar los datos y extraer las características más relevantes de las señales. Finalmente, se pueden utilizar diferentes técnicas de reconocimiento de patrones. Se pueden usar modelos supervisados o no supervisados. Se puede utilizar diferentes enfoques que sean más o menos adecuados para una aplicación determinada. Lamentablemente, no existen los pasos a seguir para determinar la estrategia más adecuada. En muchas publicaciones se utilizan métodos que no siempre funcionan bien y no son los más adecuados. También se toman algunos parámetros al azar o parámetros que, si bien se consideran más apropiados, no son los mejores. A veces los ensayos se basan solo en prueba y error. Las técnicas que carecen de una comprensión más profunda de este campo corren mayor riesgo de obtener resultados falsos, tal como lo ha demostrado Goodner, 2001.

Además, la falta de conocimiento sobre las sustancias que se pueden encontrar dificulta una correcta selección de los sensores y no se realiza un procesamiento adecuado de datos para cada una de las posibles muestras.

Debido a la necesidad de procesar datos de experimentos reales, los análisis y las investigaciones actuales son en su mayoría sobre casos muy específicos de alguna aplicación. Por lo tanto, se necesita comparar los procesos de reconocimiento de patrones, usando siempre las mismas series de datos, para poder mejorar y adaptar los algoritmos actuales. También se debería usar las mismas series de datos para poder transferir y comparar los métodos de análisis a otros laboratorios y centros de investigación.

Ejercicio 4.6

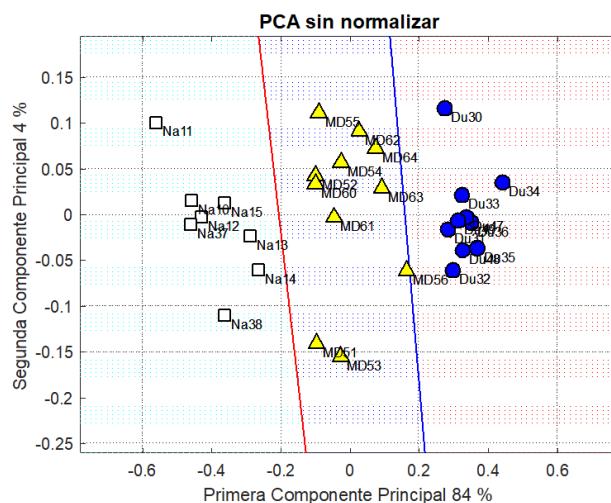
Ejemplo de Análisis de Componentes Principales (PCA)

En este ejemplo se muestra un programa en Matlab de PCA. Se dispone de un conjunto mediciones organolépticas de jugos de frutas en la variable “ratings”. Se realizaron 29 mediciones con 8 sensores de gas, por lo que la matriz ratings es de 29x8, a la cual se le agrega ruido aleatorio. En la variable “names” figuran los nombres de las mediciones, donde las 2 primeras letras corresponden al grupo medido. Mediante PCA, se reduce la dimensión y se grafica en 2 dimensiones. Luego se clasifican las distintas zonas del gráfico con Clasificadores Lineales y Cuadráticos. En la Fig. 62 se muestran los resultados sin normalizar los datos.

En la variable “curva” se obtienen las curvas de separación indicadas:

- Clasificador lineal: $'0 = -6.92447+44.7715*x+1.76654*y'$
- Clasificador cuadrático:
 $'0 = -14.6291+88.0203*x+27.8259*y+-81.0159*x^2+-71.0586*x.*y+-0.288725*y.^2'$

Resultados.



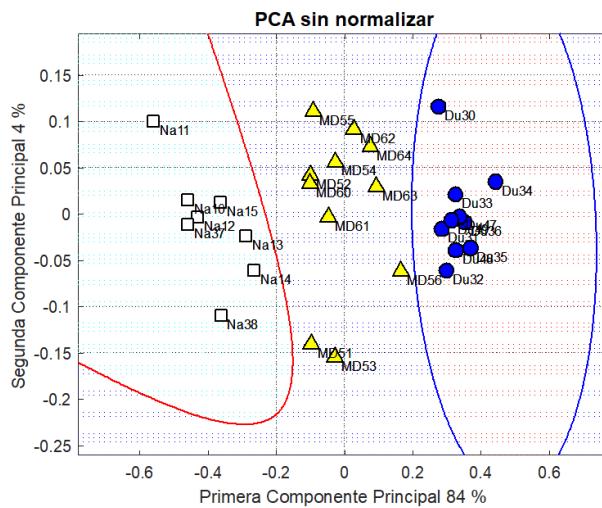


Fig. 62 – Resultados PCA de mediciones de jugos. Arriba) clasificador lineal, abajo) clasificador cuadrático

Se pide analizar dicho código

```
function Principal
% Autor: Juan Vorobioff. Reservados todos los Derechos sin previa autorización
ratings=[ -0.2688 -0.1708 -0.1738 -0.1614 -0.0573 -0.0175 -0.0940 -0.0757
-0.2247 -0.1410 -0.1420 -0.1337 -0.0457 -0.0120 -0.0677 -0.0602
-0.2004 -0.1244 -0.1245 -0.1171 -0.0387 -0.0091 -0.0562 -0.0501
-0.1853 -0.1104 -0.1140 -0.1054 -0.0333 -0.0116 -0.0508 -0.0471
-0.1774 -0.1093 -0.1124 -0.1032 -0.0311 -0.0004 -0.0500 -0.0498
-0.1743 -0.1068 -0.1085 -0.1001 -0.0328 -0.0085 -0.0428 -0.0423
-0.1673 -0.0993 -0.1035 -0.0953 -0.0290 -0.0099 -0.0451 -0.0337
-0.2037 -0.1250 -0.1284 -0.1170 -0.0369 -0.0086 -0.0562 -0.0523
-0.1897 -0.1149 -0.1203 -0.1088 -0.0356 -0.0080 -0.0554 -0.0468
-0.1701 -0.1025 -0.1061 -0.0961 -0.0255 -0.0056 -0.0499 -0.0447
-0.4395 -0.3237 -0.3282 -0.3129 -0.1388 -0.0468 -0.1988 -0.2047
-0.4517 -0.3318 -0.3342 -0.3203 -0.1477 -0.0476 -0.2144 -0.2127
-0.4181 -0.3005 -0.3008 -0.2899 -0.1239 -0.0402 -0.1824 -0.1863
-0.3873 -0.2699 -0.2740 -0.2600 -0.1095 -0.0348 -0.1563 -0.1599
-0.3717 -0.2597 -0.2648 -0.2501 -0.1023 -0.0310 -0.1473 -0.1527
-0.3437 -0.2317 -0.2354 -0.2214 -0.0864 -0.0221 -0.1281 -0.1249
-0.4231 -0.3032 -0.3033 -0.2930 -0.1282 -0.0329 -0.1793 -0.1833
-0.3835 -0.2719 -0.2700 -0.2617 -0.1071 -0.0276 -0.1573 -0.1558
-0.3644 -0.2540 -0.2505 -0.2422 -0.0977 -0.0233 -0.1421 -0.1456
-0.3470 -0.2395 -0.2388 -0.2276 -0.0873 -0.0220 -0.1306 -0.1308
-0.3286 -0.2260 -0.2279 -0.2150 -0.0769 -0.0252 -0.1210 -0.1283
-0.5904 -0.4575 -0.4625 -0.4513 -0.2742 -0.0924 -0.3613 -0.3914
-0.5752 -0.4388 -0.4424 -0.4300 -0.2527 -0.0830 -0.3390 -0.3526
-0.5353 -0.4071 -0.4104 -0.3966 -0.2158 -0.0724 -0.2996 -0.3233
-0.5204 -0.3885 -0.3911 -0.3790 -0.1962 -0.0656 -0.2775 -0.3037
-0.5047 -0.3736 -0.3775 -0.3644 -0.1820 -0.0622 -0.2660 -0.2880
-0.4956 -0.3647 -0.3686 -0.3547 -0.1731 -0.0568 -0.2534 -0.2669
-0.5755 -0.4432 -0.4442 -0.4344 -0.2435 -0.0813 -0.3244 -0.3269
-0.5310 -0.4002 -0.4038 -0.3901 -0.1947 -0.0626 -0.2745 -0.2863 ];
```

```

ruido =0.05*randn(29,8) ; %rng default
ratings = ratings +ruido; % Datos de entrenamiento + ruido
names =[ 'Du30'; 'Du31'; 'Du32'; 'Du33'; 'Du34'; 'Du35'; 'Du36'; 'Du47'; 'Du48'; 'Du49';...
'MD51'; 'MD52'; 'MD53'; 'MD54'; 'MD55'; 'MD56'; 'MD60'; 'MD61'; 'MD62'; 'MD63'; 'MD64';...
'Na10'; 'Na11'; 'Na12'; 'Na13'; 'Na14'; 'Na15'; 'Na37'; 'Na38']; % nombres de las muestras
metrica_clasificador =1 ; % métrica del clasificador
[ coefs, score] = pca_clasif( ratings, names, metrica_clasificador,100 );
metrica_clasificador =2 ;
[ coefs, score] = pca_clasif( ratings, names, metrica_clasificador,200 );
end
function [ coefs, scores] = pca_clasif(EntrenRatings, names, metrica_clasificador, fig_num )
%Generamos gráfico PCA con clasificador, Sin normalizar
figure(fig_num) ; clf(fig_num) ;
[coefs,scores,variances,t2]=PCA_princomp_jv(EntrenRatings,names, fig_num ,...
'PCA sin normalizar', 0);
% Clasificamos las distintas zonas del gráfico
Clasificar(scores,names,fig_num, metrica_clasificador ) ;
PCA_princomp_jv(EntrenRatings,names, fig_num , 'PCA sin normalizar', 0);
% Repetimos con distintas normalizaciones de datos
fig_num= fig_num+1 ; figure(fig_num) ; clf(fig_num) ;
PCA_princomp_jv(EntrenRatings,names, fig_num , 'PCA Normalizado con Desv. Estandar !', 1);
Clasificar(scores,names,fig_num, metrica_clasificador ) ;
fig_num= fig_num+1 ; figure(fig_num) ; clf(fig_num) ;
PCA_princomp_jv(EntrenRatings,names, fig_num , 'PCA Normalizado con Valores Máximos !', 2);
Clasificar(scores,names,fig_num, metrica_clasificador ) ;
fig_num= fig_num+1 ; figure(fig_num) ; clf(fig_num) ;
PCA_princomp_jv(EntrenRatings,names, fig_num , 'PCA Normalizado 3 !', 3) ;
Clasificar(scores,names,fig_num, metrica_clasificador ) ;
end
function [coefs,scores,variances,t2]=PCA_princomp_jv(ratings,names,fig_num,titulo, normalizado )
if normalizado ==0
sr = ratings ;
end
if normalizado ==1
[f c] = size(ratings) ; stdr = std(ratings);
sr = ratings./repmat(stdr,f,1);
end
if normalizado ==2
[f c] = size(ratings) ; stdr = max( abs(ratings)) ;
sr = ratings./repmat(stdr,f,1);
end
if normalizado ==3
[f c] = size(ratings) ; stdr = max(ratings); B = sort( abs(stdr) ) ;
if B(end) > 10* B(end-1)
[M,I] = max( abs(stdr) ) ; div = B(end) / B(end-1) ;
ratings(:, I) = ratings(:, I) ./repmat(div,f,1);
end
stdr = max(ratings); B = sort(stdr) ;
if B(end) > 10* B(end-1)

```

```

[M,I] = max( abs(stdr) ) ; div = B(end) / B(end-1) ;
ratings(:, I) = ratings(:, I) ./ repmat(div,f,1);
end
sr = ratings ;
end
% Calculamos PCA y Graficamos
[coefs,scores,variances,t2] = pca(sr);
figure (fig_num); title( titulo,'FontSize',12) ; hold on ;
% Separamos en clases, según los nombres
[T_real,cnames] = grp2idx(names(:,1:2)) ; T = T_real ;
ms = [ 'o' ; '^' ; 's' ; 'v' ; '>' ; 'o' ; '<' ; 's' ; 'o' ; '^' ; 'o' ; '^' ; 's' ; ...
'v' ; '>' ; 'p' ; '<' ; 'h' ; 'o' ; '^' ; 'o' ; '^' ; 's' ; 'v' ; '>' ; 'p' ; '<' ; ...
'h' ; 'o' ; '^' ] ; % Marker Specifiers
mfc = [ 'b' ; 'y' ; 'w' ; 'g' ; 'c' ; 'k' ; 'm' ; 'w' ; 'w' ; 'w' ; 'r' ; 'y' ; 'b' ; 'g' ; ...
'c' ; 'k' ; 'm' ; 'w' ; 'w' ; 'r' ; 'y' ; 'b' ; 'g' ; 'c' ; 'k' ; 'm' ; 'w' ; 'w' ; 'w' ] ; % MarkerFaceColor
mec = [ 'k' ; ...
'k' ; 'k' ] ; % 'MarkerEdgeColor'
for jj=1 : max(T)
    plot(scores(find(T==jj),1),scores(find(T==jj),2), ms(jj) , 'MarkerSize', 9 , 'LineWidth',1, ...
        'MarkerFaceColor',mfc(jj) , 'MarkerEdgeColor',mec(jj) ) ; %, names(i,1:2)
end
for i=1: size(scores,1)
    text( scores(i,1),scores(i,2),{' ' ; [ ' ' names(i,:) ] } , 'FontSize', 8 ) ;
end
percent_explained = 100*variances/sum(variances) ;
s1=['Primera Componente Principal ' int2str(percent_explained(1)) '%' ] ;
s2=['Segunda Componente Principal ' int2str(percent_explained(2)) '%' ] ;
xlabel(s1) ; ylabel(s2) ; grid on ; box on ;
end
function []=Clasificar(scores,names,fig_num, metrica_cl_p )
[T_real,cnames] = grp2idx(names(:,1:2)) ;
% Clasificamos
metrica_cl_array = {'linear' ; 'quadratic' ; 'diaglinear' ; 'mahalanobis' ; 'diagquadratic'} ;
ss=get(fig_num) ; % axis(ss.CurrentAxes) ;
axis_act = [ get(ss.CurrentAxes,'XLim') get(ss.CurrentAxes,'YLim') ] ;
f= 1.3 ; xlim([axis_act(1)*f axis_act(2)*f]) ; ylim([axis_act(3)*f axis_act(4)*f]) ;
axis_act = [ get(ss.CurrentAxes,'XLim') get(ss.CurrentAxes,'YLim') ] ;
% Generamos una grilla y la clasificamos
[X1,Y1] = meshgrid(linspace(axis_act(1),axis_act(2)),linspace(axis_act(3),axis_act(4)) );
X = X1(:); Y = Y1(:);
metrica_cl = cell2mat(metrica_cl_array(metrica_cl_p)) ;
[C,error1,posterior,logp,coeff] = classify([X Y], scores(:,1:2),T_real, metrica_cl );
%Visualizamos la clasificación
hold on; gscatter( X,Y,C,'rbcy',1,1,'off');
switch metrica_cl
    case {'linear' , 'diaglinear' }
        K = coeff(1,2).const; L = coeff(1,2).linear;
        curva = sprintf('0 = %g+%g*x+%g*y', K,L(1),L(2));
        h1 = ezplot(curva, axis_act); set(h1,'Color','b','LineWidth',1)
        if( max(T_real) >= 3 ) % ==3

```

```

K = coeff(2,3).const; L = coeff(2,3).linear;
curva = sprintf('O = %g+%g*x+%g*y', K,L(1),L(2));
h1 = ezplot(curva,axis_act); set(h1,'Color','r','LineWidth',1)
end
if( max(T_real) >= 4 )  %% ==4
    K = coeff(3,4).const; L = coeff(3,4).linear;
    curva = sprintf('O = %g+%g*x+%g*y', K,L(1),L(2));
    h1 = ezplot(curva,axis_act); set(h1,'Color','c','LineWidth',1)
end
case { 'mahalanobis' 'quadratic' 'diagquadratic' }
    K = coeff(1,2).const; L = coeff(1,2).linear;
    Q = coeff(1,2).quadratic ;
    curva = sprintf('O = %g+%g*x+%g*y+%g*x^2+%g*x.*y+%g*y.^2',...
        K,L,Q(1,1),Q(1,2)+Q(2,1),Q(2,2));
    h2 = ezplot(curva,axis_act); set(h2,'Color','b','LineWidth',1)
    if( max(T_real) >= 3 )  %% ==3
        K = coeff(2,3).const ; % coeficientes constantes
        L = coeff(2,3).linear;
        Q = coeff(2,3).quadratic ;
        curva = sprintf('O = %g+%g*x+%g*y+%g*x^2+%g*x.*y+%g*y.^2',...
            K,L,Q(1,1),Q(1,2)+Q(2,1),Q(2,2));
        h2 = ezplot(curva,axis_act); set(h2,'Color','r','LineWidth',1)
    end
    if( max(T_real) >= 4 )
        K = coeff(3,4).const; L = coeff(3,4).linear; % coeficientes
        Q = coeff( 3,4 ).quadratic ;
        curva = sprintf('O = %g+%g*x+%g*y+%g*x^2+%g*x.*y+%g*y.^2',...
            K, L, Q(1,1), Q(1,2)+Q(2,1),Q(2,2));
        h2 = ezplot(curva,axis_act); set(h2,'Color','c','LineWidth',1)
    end
end
axis( axis_act ); title ([ 'PCA - Clasificador: ' metrica_cl ]); box on ;
end

```

Estimación de la función de densidad de probabilidad

Mediante la función de densidad de probabilidad obtenemos la relación que tienen las observaciones con su probabilidad. Teniendo en cuenta determinadas mediciones, algunos resultados de las variables aleatorias tendrán densidad de probabilidad baja y otros resultados densidad de probabilidad alta. La forma que tiene la densidad de probabilidad se denomina distribución de probabilidad. Se puede utilizar la función de densidad de probabilidad para calcular algunos valores específicos de una variable aleatoria. Mediante la función de densidad de probabilidad se puede conocer si una determinada observación es poco probable y se puede considerar como una anomalía, y poder eliminarse. También se utiliza para elegir los algoritmos de aprendizaje apropiados basados en la distribución de probabilidad (The Mathworks, 2019).

Para observaciones aleatorias, la densidad de probabilidad se puede aproximar a través de un proceso de estimación de densidad de probabilidad. La estimación de densidad se basa en datos observados, se estima una función de densidad de probabilidad no observable. Se considera una

gran población o número de muestras, donde los datos estudiados se consideran como una muestra aleatoria tomada de la población grande.

Mediante suposiciones se pueden obtener distribuciones de probabilidad en forma teórica. Las distribuciones asignan valores de probabilidad que una variable aleatoria este comprendido en un intervalo o bien tome algún valor un valor específico.

Se utilizan distintas técnicas para estimar la densidad. La forma más básica para estimar la densidad es mediante un histograma escalado.

Dada una variable aleatoria llamada x , su función de distribución de probabilidad es $p(x)$. La relación entre los resultados de la variable aleatoria x y su probabilidad se conoce como densidad de probabilidad o simplemente densidad. La distribución de probabilidad puede tener distintas formas, puede ser uniforme, normal, exponencial, etc.

Debemos estimar la distribución de probabilidad de una variable aleatoria, ya que generalmente no tenemos acceso a todos los resultados posibles que puede tomar una variable aleatoria. Solo tenemos acceso a un conjunto de observaciones. Hallar esta distribución, se conoce como estimación de densidad de probabilidad.

Se utilizan histogramas para tratar de identificar una distribución de probabilidad, también se ajustan modelos para estimar la distribución.

La función de densidad de probabilidad (o PDF) normal está dada por:

$$y = f(x/\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}; \text{ para } x \in \mathbb{R} \quad (4.18)$$

La PDF o función de densidad de probabilidad de Rayleigh resulta:

$$y = f(x/b) = \frac{x}{b^2} \cdot e^{-\frac{x^2}{b^2}} \quad (4.19)$$

Se pueden usar distintas funciones de densidad de probabilidad, por ejemplo, exponencial, normal, Rayleigh, geométrica, Chi cuadrado, etc.

Ejercicio 4.7

Gráficos de Funciones de densidad de probabilidad

Se grafican algunas PDF en Matlab® y se muestran los resultados en la Fig. 63.

```
% Vector de entrada
x = -5:.05:5;
mu = 0 ; sigma = 1;
y_normal = pdf( 'Normal', x, mu, sigma )
y_Rayleigh = raylpdf(x,1); %plot(x,y,'+')
y_chi = chi2pdf(x,2.5);
% Graficamos pdf
plot(x, y_normal,'LineWidth',2); hold on
plot(x, y_Rayleigh,'LineWidth',2);
```

```
plot(x, y_chi,'LineWidth',2);
legend('PDF Normal','PDF Rayleigh','PDF Chi cuadrado'); grid on;
```

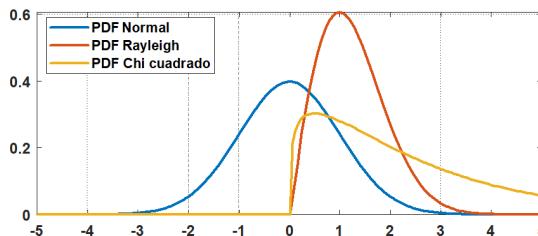


Fig. 63 – Función de densidad de probabilidad (PDF) Normal, Rayleigh y Chi cuadrado

La función de distribución acumulada (función de probabilidad acumulada) correspondiente a una variable aleatoria real X , es una función que describe la probabilidad de que la variable X tenga un valor menor o igual que x .

Para el caso discreto, dada X una variable aleatoria discreta que tiene una función de probabilidad $f(x)$, su función de probabilidad acumulada se calcula de la siguiente forma:

$$F(x) = P(X \leq x) = \sum_{u < x} f(u) \quad (4.20)$$

Para el caso continuo, dada X una variable aleatoria y continua con función de probabilidad $f(x)$, su función de probabilidad acumulada se calcula de la siguiente forma:

$$F(x) = P(X \leq x) = \int_{-\infty}^x f(u) \quad (4.21)$$

Ejercicio 4.8

Gráficos de Funciones de densidad de probabilidad

Se analizan los pacientes de un hospital con cierta enfermedad. Cada uno tiene una probabilidad de recuperarse de 0.3, con densidad de probabilidad binomial. Habiendo 20 pacientes, se pide:

a) Calcular la probabilidad de que al menos 10 personas sobrevivan.

Resolución

$X = 20$ es el número de supervivientes.

Podemos usar estas funciones de Matlab®

```
binopdf( x, n, p )
binocdf( x, n, p )
binocdf( 9, 20, 0.3 )
ans = 0.952
```

$$P(X \geq 10) = 1 - P(X < 10) = 1 - \sum_{x=0}^9 b(x; 20, 0.3) = 1 - 0,952 = 0.048 = 4,8 \%$$

b) Calcular la probabilidad expresada en porcentaje de que sobrevivan entre 3 y 8 personas

`binocdf(8, 20, 0.3)`

`ans = 0.8867`

`binocdf(2, 20, 0.3)`

`ans = 0.0355`

$$P(3 \leq X \leq 8) = \sum_{x=3}^8 b(x; 20, 0.3) = \sum_{x=0}^8 b(x; 20, 0.3) - \sum_{x=0}^2 b(x; 20, 0.3) = 0,8867 - 0,0355 \\ = 0,8512 = 85,12 \%$$

c) Probabilidad de supervivencia para exactamente 5 personas

`binocdf(5, 20, 0.3)`

`ans = 0.4164`

`binocdf(4, 20, 0.3)`

`ans = 0.2375`

$$P(X = 5) = \sum_{x=0}^5 b(x; 20, 0.3) - \sum_{x=0}^4 b(x; 30, 0.3) = 0,4164 - 0,2375 = 0,1789 = 17,89 \%$$

Utilizando la PDF resulta más simple:

`binopdf(5, 20, 0.3)`

`ans = 0.1789`

$$P(X = 5) = b(5; 20, 0.3) = 0,1789 = 17,89 \%$$

Ejercicio 4.9

Ejemplo de Estimación de distribución de probabilidad

Se pide estimar la distribución de probabilidad mediante algoritmo de estimación de máxima verosimilitud. Se utiliza la función `fitdist` de Matlab® que se adapta a la mayoría de las distribuciones. Analizar el siguiente código, en la Fig. 64 se muestran los resultados.

```
% Importamos los datos
load examgrades
% Los datos contienen matriz de 120x5, correspondiente a 120 alumnos y notas.
% Las notas tienen escala de 0 a 100.
% Tomamos la primera columna de notas
x = grades(:,1);
% Ajustamos una distribución normal, utilizando fitdist para crear un objeto de distribución
```

```
% de probabilidad
% Utilizamos distribución normal.
% La función fitdist devuelve los parámetros estimados mu y sigma de la distribución normal
pd = fitdist(x,'Normal')
% Intervalos de confianza del 95% para los parámetros de distribución.
% Calculamos la media para las calificaciones de los estudiantes
% El valor medio de los exámenes es igual al parámetro mu estimado por fitdist.
m = mean(pd)
pd.mu
% Graficamos histograma. Y comparamos visualmente la distribución normal ajustada con las
calificaciones reales
x_pdf = [1:0.1:100];
y = pdf(pd,x_pdf);
figure ; histogram(x,'Normalization','pdf')
line(x_pdf,y, 'linewidth',3, 'color', 'r')
xlabel('Calificaciones'); grid on
% El pdf para la distribución ajustada tiene la misma forma que el histograma correspondiente a las
% calificaciones.
% Utilizamos la función de distribución acumulada inversa (icdf) para determinar el límite para el 10
% por ciento superior de las notas. Este límite es igual al valor en el que la CDF de la distribución
% de probabilidad es igual a 0.9.
A = icdf( pd, 0.9 )
% Analizando la distribución obtenida, el 10 por ciento de los alumnos recibió una nota superior a
% 86.1837. De manera equivalente, el 90 por ciento de los estudiantes recibió una calificación menor
% o igual a 86.1837.
% Guardamos el objeto pd
%save myobject.mat pd
```

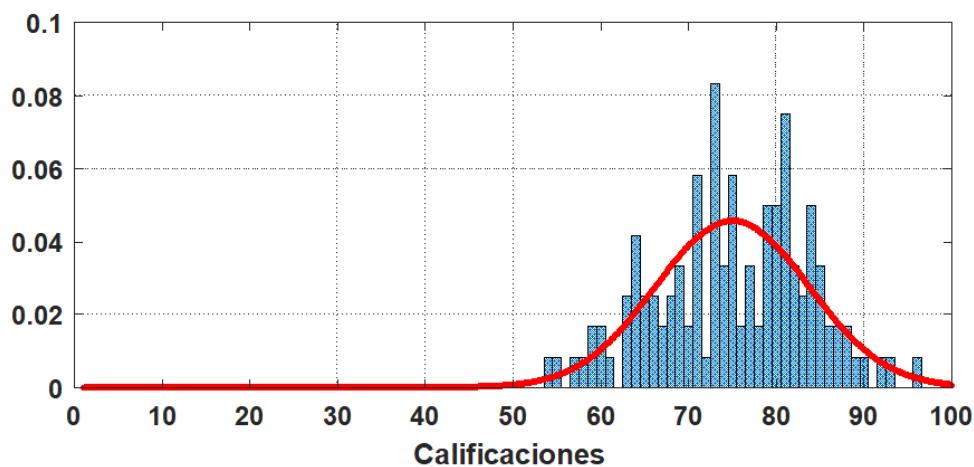


Fig. 64 – Ejemplo de estimación de densidad de probabilidad normal

Preprocesamiento, normalización y extracción de características o parámetros

En un sistema de medición, la adquisición de las señales es el primer paso para analizar los datos; se obtiene esta información proveniente de los sensores mediante señales eléctricas. Muchas veces este paso causa mucha dificultad en la clasificación debido a las características y limitaciones que tiene el transductor, puede limitar o falsear la información. Como salida se obtiene un vector de datos patrón, en un espacio patrón. Este vector patrón a la segunda etapa, donde se extraen características o parámetros. La extracción de parámetros utiliza una o más transformaciones de la matriz de datos de entrada, para generar una nueva matriz de datos más relevantes. Esta extracción de parámetros se considera como un proceso de reducción de dimensión; los datos se convierten del espacio de patrones a un espacio nuevo de parámetros. Donde los parámetros deberían ser fácilmente evaluados. Esto se logra de dos maneras distintas, la primera mediante un claro entendimiento físico y la segunda mediante el análisis de parámetros sin tener un claro entendimiento físico. Todo este procesamiento previo de datos afecta al clasificador utilizado en la próxima etapa.

Como ejemplos de extracción de características de un conjunto de señales, se pueden tomar los valores máximos, las integrales en cierta zona y los valores de las señales en ciertos puntos. Si se tiene un conjunto de imágenes, por ejemplo, se puede tomar la presencia o ausencia de objetos en la imagen, o el resultado de máscaras o filtros aplicados a las imágenes.

En muchos casos resulta necesario normalizar las mediciones. Por ejemplo, al medir concentraciones de muestras se puede generar un mal escalado en los patrones. Los algoritmos de reconocimiento de patrones generalmente examinan las diferencias entre diferentes patrones, pero un mal escalado puede tapar y enmascarar estas diferencias. La normalización se utiliza para remover estos efectos no deseados. La Tabla VII muestra distintas técnicas de normalización (Del Brío, 2007).

Tabla VII - Técnicas estándar de Normalización

Normalización	Ecuación	
Escala Relativa 1	$X_{ij} = \frac{X_{ij}}{\max(X)}$	(4.22)
Escala Relativa 2	$X_{ij} = \frac{X_{ij}}{\max(X_j)}$	(4.23)
Escala Relativa 3	$X_{ij} = \frac{X_{ij}}{\max(X_r)}, X_r \text{ es una señal de referencia}$	(4.24)
Escala Relativa 4	$X_{ij} = \frac{X_{ij}}{\ X_i\ }$	(4.25)
Extracción de señal de fondo (respuesta en blanco)	$X_i = X_i - X_b, \text{ donde } X_b \text{ es la respuesta a la señal de blanco}$	(4.26)
Promedio de la señal	$X_{ij} = \frac{1}{N} \sum_{k=1}^N X_{ij}^k, N \text{ repeticiones}$	(4.27)
Auto escala	$X_{ij} = \frac{X_{ij} - X_j}{\sigma x_j}$	(4.28)

Rango de escala 1	$X_{ij} = \frac{X_{ij} - \min(X_j)}{\max(X_j) - \min(X_j)}$	(4.29)
Rango de escala 2	$X_{ij} = \frac{2(X_{ij} - \min(X_j))}{\max(X_j) - \min(X_j)} - 1$	(4.30)
Extracción de línea de base	$X_{ij} = X_{ij} - X_{1j}$	(4.31)

X : matriz de parámetros con n muestras con p sensores. Se tiene X_{ij} donde i indica el numero de muestra y j es el número de sensor.

X_j contiene todas las n respuestas del sensor j ,

X_i contiene todas las p respuestas de la muestra i .

Mediante la escala relativa 1, ver ecuación (4.22), tenemos un escalamiento global de todos los datos normalizando con un valor máximo de 1.

En la escala relativa 2, ecuación (4.23), normaliza los valores asignando el valor máximo 1 a cada uno de los sensores.

En la escala relativa 3, ecuación (4.24), se normaliza respecto a una referencia.

En la escala relativa 4, ecuación (4.25) se utiliza la distancia Euclíadiana normalizada, es un método considera local. Se utiliza para compensar las diferencias de concentraciones entre distintas muestras. Para mediciones cualitativas, frecuentemente se utiliza normalización por escala relativa.

Para suavizar el ruido, se utiliza la normalización por señal media, ecuación (4.27), se calcula el promedio de varias mediciones.

La técnica de centrado al valor medio centra todos los datos en el origen.

En la normalización por auto escala, ver ecuación (4.28), se centran los datos en el origen y se asigna desviación estándar unitaria. Se considera un método global y se utiliza cuando se tienen mediciones de magnitudes diferentes.

Las normalizaciones con rango de escala 1 y con rango de escala 2, ver ecuaciones (4.29) y (4.30), establecen los límites de los datos de entrada entre 0 y 1 y entre -1 y 1, respectivamente. Estas normalizaciones generalmente se utilizan en redes neuronales.

La normalización realizada por resta de línea de base (background), ver ecuación (4.26), intenta corregir el ruido de los datos restando la respuesta de un blanco.

En la resta de línea de base, ver ecuación (4.31), se remueve la línea de base que presenta cada sensor y se utiliza muy frecuentemente en datos temporales. Las aplicaciones con datos temporales producen un número grande de mediciones. Estos datos se suelen reducir en la etapa previa al reconocimiento de patrones.

Exploración de datos mediante métodos gráficos

En muchos casos se tienen grandes volúmenes de datos de alta dimensión. Generalmente es un gran desafío poder extraer información útil para resolver el problema estudiado. Mediante los métodos gráficos se analizan los datos en forma simple. Algunos métodos utilizan gráficos con dimensiones altas, mientras otros métodos reducen los datos a dos o tres dimensiones para su visualización y análisis en forma sencilla.

Gráfico de barras: se visualizan los datos en forma simple y útil. Se grafica las respuestas una al lado de otra, en forma horizontal o vertical, con un ancho pequeño. Se puede graficar en forma individual o mediante el promedio de varias muestras.

Gráficos polares: se muestran datos de altas dimensiones en gráficos simples circulares de dos dimensiones. Para cada variable se grafican ejes desde el origen, todos los ejes tienen el mismo ángulo entre sí. En cada variable se grafica la magnitud en su eje correspondiente y luego se unen los puntos correspondientes a los distintos ejes. Si se tienen diferencias pequeñas, se suelen procesar los datos obtener mejor separación. Por ejemplo, al vector de datos se le puede restar una referencia y escalarlo de manera de enfatizar las diferencias.

A modo de ejemplo, en la Fig. 58, se muestra la respuesta de 32 sensores de gas Poliméricos de Conducción. Se miden 2 grupos de aceites de oliva especiales de distintas marcas, (GV) e (IN). Para cada grupo se calculan los valores medios y su desviación estándar. Luego para mejorar la visualización, los datos de cada sensor se normalizan respecto del grupo de menor valor medio (en este caso GV). Por último, se grafican todos los valores medios normalizados y sus desviaciones estándar. En la figura, se puede observar que solo algunos sensores resultan más selectivos que otros, brindando más información para separar los grupos de aceites.

Mediante este análisis se pueden descartar los sensores que no sirvan y no brinden información para el procesamiento de datos.

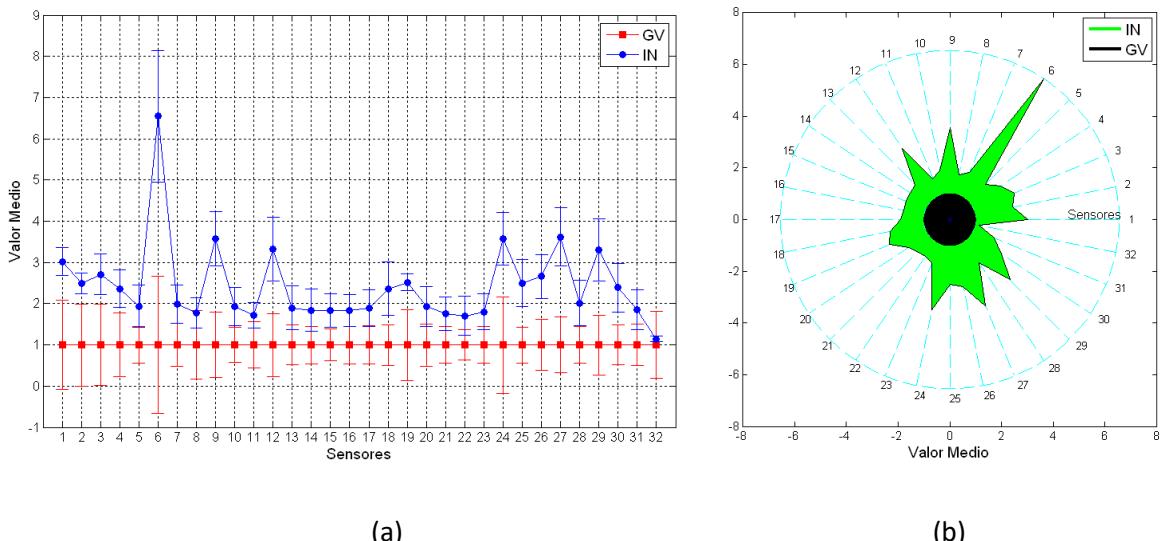


Fig. 65 – Valor medio de los sensores para muestras de aceites. Aceite (GV) y aceite (IN). Los valores se normalizan respecto de GV. (a) Gráfico de valores medios con desviación estándar. (b) Gráfico Polar con valores medios.

Aprendizaje y generalización

La generalización de una red neuronal busca tener una buena respuesta de la red con entradas nuevas que nunca se usaron. Estas entradas nuevas se llaman de **testeo** y no se utilizan para entrenar la red, es decir no se utilizan para ajustar los parámetros internos de la red.

Las redes neuronales se pueden implementar como clasificadores o como regresores. En el caso de clasificadores se asigna una clase discreta a un vector de entradas. En el caso de regresores, se asigna un vector de salida continuo o analógico a un vector de entrada continuo.

Las funciones de clasificación y regresión estimadas con las redes neuronales son desconocidas. Mediante un conjunto de entrenamiento, se brindan ejemplos de entrada-salida de la función. En el entrenamiento de las redes neuronales, se identifica esta "función desconocida", conociendo solamente los datos de entrenamiento. Se estiman los parámetros de la función, es decir los pesos y los sesgos de las neuronas, para que replique lo mejor posible la relación que tienen las entradas y salidas del entrenamiento. Y también se espera que generalice bien a nuevos datos. Resulta que obtener el mejor rendimiento de generalización en datos nuevos no suele ser la misma solución que replicar los datos de entrenamiento de la mejor manera posible.

Por ejemplo, si tenemos muy pocos patrones de entrenamiento, pero una red neuronal grande, será fácil hallar los pesos para reproducir el conjunto de entrenamiento, pero no parece probable que la red resultante haya aprendido correctamente para "testear" nuevos datos. Por otro lado, si tenemos muchos patrones de entrenamiento y logramos entrenar a la red para replicarlos, entonces parecería que es más probable que responda correctamente a datos nuevos. Estas intuiciones deben ser precisas, hay que utilizar métodos para mejorar la generalización de las redes neuronales.

Evaluación de la generalización: conjuntos de entrenamiento, prueba y validación

Se puede estimar la generalización de una red probando su rendimiento con datos nuevos. Sin embargo, debemos tener cuidado con los datos que usamos; si seguimos usando el mismo conjunto de datos de entrenamiento, aunque el algoritmo de entrenamiento no esté utilizando estos datos, estamos tratando de obtener el mejor rendimiento en este conjunto en particular. Normalmente se trabaja con tres conjuntos de datos: Conjunto de entrenamiento: estos son los datos los utiliza el algoritmo de entrenamiento para ajustar los pesos w y los sesgos b de la red.

Conjunto de validación (desarrollo): estos datos se utilizan durante el entrenamiento para evaluar el rendimiento actual de la red. Estos datos se pueden usar para guiar el entrenamiento, por ejemplo, controlar la tasa de aprendizaje, decidir cuándo detenerse, seleccionar distintas redes, etc.

Conjunto de prueba (evaluación o testeo): Una vez entrenada la red, estos son los datos de prueba desconocidos que queremos hallar su respuesta. Estos datos no influyen en el entrenamiento.

Entrenamiento y generalización

En la etapa entrenamiento, una red neuronal con determinadas arquitecturas puede implementar un conjunto de funciones ajustando sus parámetros.

En el reconocimiento de patrones la red no siempre es capaz de aprender correctamente todos los datos de entrenamiento; de hecho, no siempre es deseable hacerlo. La generalización es una medida que nos dice qué tan bien se desempeña la red en el problema real una vez que se completa el entrenamiento. Esto se puede medir observando el desempeño de la red en los datos de evaluación no utilizados durante el proceso de entrenamiento. Por definición, estos datos no están disponibles cuando se entrena la red.

Dado que maximizar el rendimiento de generalización de la red implica producir el mejor rendimiento posible en nuevos datos, un enfoque simple para estimar el rendimiento de generalización de las redes es evaluar su rendimiento en un conjunto de validación. Se entrena la red de forma habitual, minimizando la función de error respecto al conjunto de datos de entrenamiento. El rendimiento en un conjunto de entrenamiento se estima mediante el conjunto de validación.

Por ejemplo, si estimamos un pequeño conjunto de datos de entrenamiento usando dos redes, una red simple implementando una línea recta (correspondiente a una red de una sola capa) y otra red con muchas capas más compleja con muchas unidades ocultas. La red más compleja es capaz de aproximar la función exactamente, mientras que la red de una sola capa ajusta los datos razonablemente bien con su línea recta, pero con mayor error.

Para los datos de entrenamiento la red compleja responde bien. Sin embargo, no generaliza bien para datos nuevos. La red simple con una recta puede representar bien los datos nuevos de testeo.

Como conclusiones podemos decir en forma general lo siguiente:

- a) Un buen desempeño en los datos de entrenamiento no necesariamente conduce a un buen desempeño en generalización;
- b) Es más probable que las soluciones simples generalicen mejor que las soluciones complejas (a menos que una gran cantidad de datos de entrenamiento indique lo contrario).
- c) Las redes complejas necesitan mayor cantidad de datos de entrenamiento.
- d) Un modelo demasiado simple resulta poco flexible y tiene sesgos grandes.
- e) Un modelo complejo, tiene mayor flexibilidad en relación con los datos de entrenamiento y tiene sesgos pequeños.

Métodos de validación

No importa cuán poderoso sea un modelo de aprendizaje automático y / o aprendizaje profundo, nunca puede hacer lo que queremos que haga con datos incorrectos. Puede contener ruido aleatorio (es decir, puntos de datos que dificultan ver un patrón), baja frecuencia de una determinada variable categórica, baja frecuencia de la categoría objetivo (si la variable objetivo es categórica) y valores numéricos incorrectos, etc. son solo algunas de las formas en que los datos pueden estropear un modelo.

Si bien el proceso de validación no puede encontrar directamente qué está mal, pero puede mostrarnos a veces que hay un problema con la estabilidad del modelo.

Un modelo es robusto si su variable o etiqueta de salida, es precisa incluso en el caso que una o más de variables independientes de entrada, o características, cambian drásticamente sus valores debido a circunstancias imprevistas.

En la práctica, un modelo de aprendizaje automático nunca resulta 100% exacto teniendo en cuenta todos los sesgos, variaciones y errores irreducibles. Es inevitable el error irreducible, ya que no se puede tener un modelo perfecto (en la práctica). Sin embargo, se considera un buen modelo aquel que puede agregar información en la vida real.

Análisis de sensibilidad de la red

Mediante el análisis de sensibilidad se determina cómo se ve afectada la salida en función de las variaciones de los datos debido al ruido o fluctuaciones. El análisis de sensibilidad le permitirá explorar la generalización de los límites de decisión de su modelo para ver realmente el impacto de la falta de generalización.

Validación cruzada

La validación cruzada es una técnica para evaluar el rendimiento que tiene un modelo de predicción estadística en un conjunto de datos independientes. El objetivo es asegurarse de que el modelo funcione correctamente. La validación cruzada se lleva a cabo durante el entrenamiento, donde el usuario evaluará si el modelo es propenso a fallar o sobreajustarse a los datos. Los datos que se utilizarán para la validación cruzada deben tener la misma distribución que la variable de testeo. En la Fig. 66 se muestra ejemplo de validación cruzada para una sola iteración, donde se separan los datos en 20 % para prueba y el resto se entrena el modelo.



Fig. 66 – Validación cruzada para única iteración

Se describen diferentes técnicas de validación cruzada

- La validación cruzada k-fold se utiliza cuando se quiere conservar la mayor cantidad de datos posible para la etapa de entrenamiento y no correr el riesgo de perder datos valiosos en el conjunto de validación. El conjunto de datos se divide en k número de conjuntos en los que se utilizará un conjunto como datos de validación. El resto se utilizará como conjunto de datos de entrenamiento. Esto se repetirá el número de veces que lo especifique el usuario.

En el caso de regresión, se analizarán los resultados con el promedio de los errores, por ejemplo, error cuadrático medio. En la clasificación, se tomará como resultado final el promedio de los resultados, es decir precisión, tasa de verdaderos positivos, etc.

En la Fig. 67 se muestra un ejemplo de validación cruzada k-fold

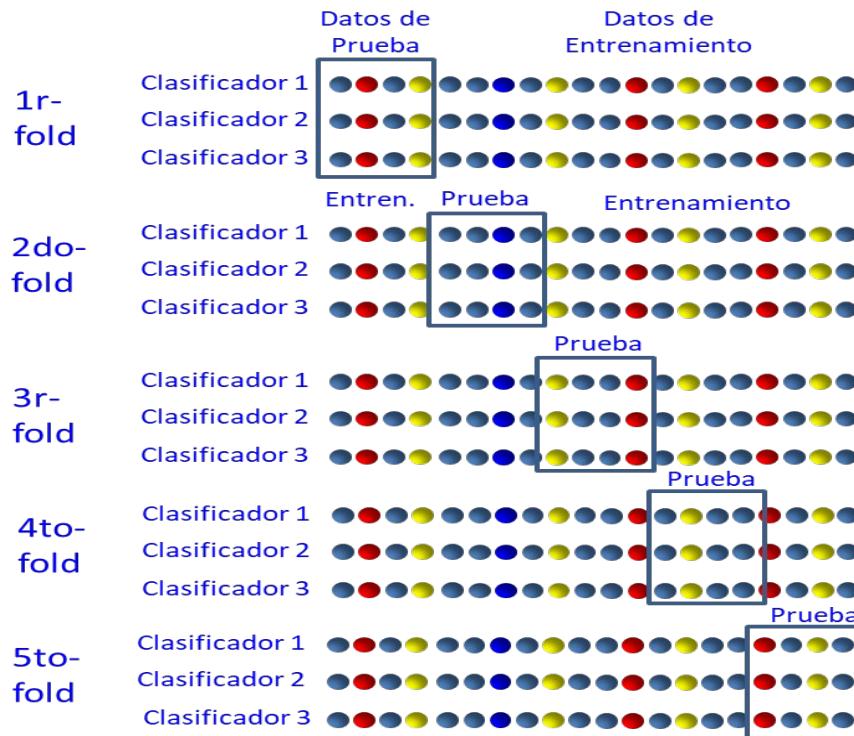


Fig. 67 – Validación cruzada k-fold con $k=5$ y 3 clasificadores

- b) El método de retención es el método de validación cruzada más simple. Dividimos el conjunto de entrenamiento y sacamos una pequeña parte para utilizar como conjunto de validación. Entrenamos nuestro modelo con el conjunto de entrenamiento nuevo y más pequeño, y validamos la precisión del modelo con el conjunto de validación, que aún no ha sido visto por el modelo.
- c) Validación de dejar uno fuera (Leave-One-Out cross validation: LOOCV). Esta validación Leave-One-Out es similar a la validación cruzada de k-fold, pero se extrae solo un dato por vez. La iteración se lleva a cabo n veces y el conjunto de datos se dividirá en $n-1$ conjuntos de datos, el dato extraído se utiliza como dato prueba. El rendimiento se mide de la misma manera que la validación cruzada de k-fold.

La validación de un conjunto de datos brinda confianza respecto a la estabilidad del modelo. Mediante la validación se optimiza el rendimiento y se puede mantener estable el modelo en un período de tiempo antes de tener que volver a entrenarse.

Sobrecargar y subestimar son los dos errores más comunes en la construcción de los modelos.

Matriz de Confusión

La matriz de confusión mide qué tan bien se ha hecho la clasificación. Es una medida de rendimiento en la clasificación en los métodos de aprendizaje automático, incluidas las ANN. Se busca mejorar la efectividad y el rendimiento. En la Fig. 68 se muestra la estructura de la matriz de confusión. Se detalla el significado de los 4 elementos de la matriz:

VP: Verdadero positivo, se predijo positivo y es cierto.

VN: Verdadero Negativo, se predijo negativo y es cierto.

FP: Falso Positivo, error Tipo 1 donde se predijo positivo y es falso.

FN: Falso Negativo, error Tipo 2 donde se predijo negativo y es falso.

		Valores actuales	
		Positivos (1)	Negativos (0)
Valores Predecidos	Positivos (1)	VP	FP
	Negativos (0)	FN	VN

Fig. 68 – Estructura de la matriz de confusión

Se describen varias métricas de evaluación de la matriz

Sensibilidad (Recall): Se analizan todas las clases positivas predecidas correctamente respecto de las positivas actuales o reales. Debe ser lo más alto posible.

$$\text{Recall} = \frac{VP}{VP + FN}$$

Precisión: Se analizan todas las clases positivas predecidas y se analizan cuántas son realmente positivas.

$$\text{Precisión} = \frac{VP}{VP + FP}$$

Exactitud: De todas las clases, cuántas se predijeron correctamente. Tiene que ser lo más alto posible.

$$\text{Exactitud} = \frac{VP + VN}{Total}$$

Puntuación-F o Medida-F: es difícil comparar dos modelos que tienen baja precisión y/o alta sensibilidad o viceversa. Para poder comparar 2 modelos, usamos F-Score.

$$\text{Medida - F} = \frac{2 * \text{Recall} * \text{Precisión}}{\text{Recall} + \text{Precisión}}$$

Ejercicio 4.10**Matriz de confusión en Matlab con 2 clases**

Se tiene un conjunto de datos separado en 2 grupos. Se identifican los grupos con el siguiente vector:

`targets = [1 2 1 1 1 2 1 1 1 1 1 2 2 2 2 2 2]'; % Grupos verdaderos`

Se predicen los grupos obteniendo el siguiente vector:

`outputs = [1 2 1 1 1 1 1 1 2 2 2 2 2 2 2 2]'; % Grupos predecidos`

Se pide hallar la matriz de confusión y calcular sus métricas de evaluación

Resolución analítica

En base a la Fig. 68 y a los datos del enunciado se arma la siguiente matriz de confusión:

8	2
1	9

$$VP = 8 ; VN = 9 ; FP = 2 ; FN = 1 ;$$

$$\text{Exactitud} = \frac{VP+VN}{Total} = \frac{8+9}{8+9+2+1} = 0,85$$

$$\text{Precisión} = \frac{VP}{VP+FP} = \frac{8}{8+2} = 0,8$$

$$\text{Recall} = \frac{VP}{VP+FN} = \frac{8}{8+1} = 0,889$$

$$\text{Medida - F} = \frac{2*\text{Recall}*\text{Precisión}}{\text{Recall}+\text{Precisión}} = \frac{2*0,889*0,8}{0,889+0,8} = 0,842$$

Resolución en Matlab

`targets = [1 2 1 1 1 2 1 1 1 1 1 2 2 2 2 2 2]'; % Grupos verdaderos`

`outputs = [1 2 1 1 1 1 1 1 2 2 2 2 2 2 2]'; % Grupos predecidos`

`% Matriz de confusión y evaluación`

`confMatr = confusionmat(targets, outputs)`

`Exactitud = 100*sum(diag(confMatr))/sum(confMatr(:))`

`Precision = 100*confMatr(1,1)./sum(confMatr(1,:))`

`Recall = 100*confMatr(1,1)./sum(confMatr(:,1))`

`Medida_F = 2*Recall*Precision /(Recall +Precision)`

`cm = confusionchart(targets, outputs) % Introducido en Matlab 2018b`

Los resultados coinciden con la resolución analítica. En la Fig. 69 se muestra la matriz de confusión obtenida.

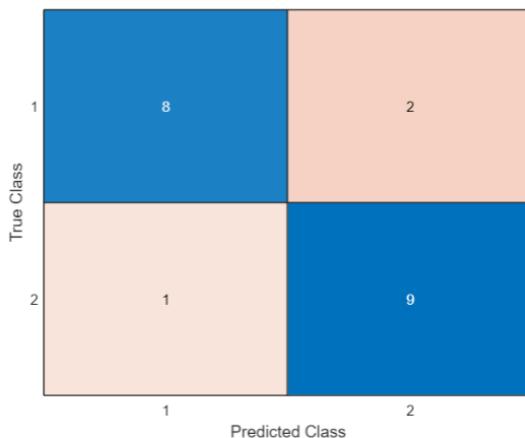


Fig. 69 – Matriz de confusión con 2 clases calculada con Matlab

Ejercicio 4.11

Matriz de confusión en Matlab con 4 clases

Analizar el siguiente programa. Se muestran los resultados en la Fig. 70.

```

g1 = [ 4 4 4 4 4 4 3 3 2 2 2 1 1 ]'; % Grupos verdaderos
g2 = [ 4 4 4 4 4 4 1 3 2 2 4 1 1 ]'; % Grupos predecidos
C = confusionmat(g1,g2)
confusionchart(C) % Matlab 2018b

```

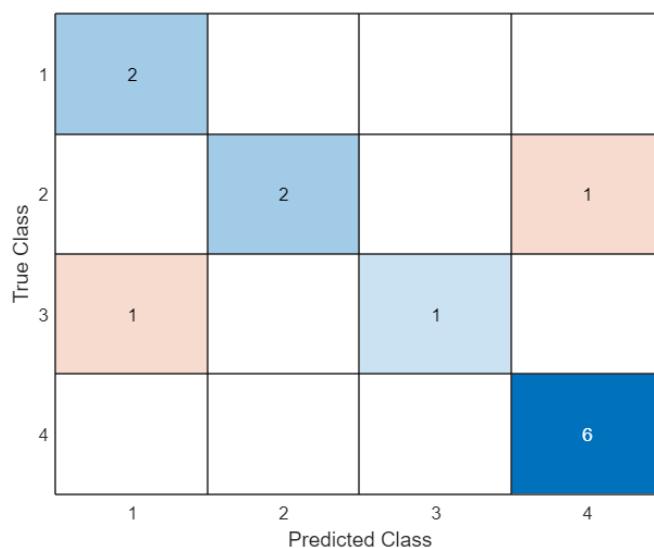


Fig. 70 – Matriz de confusión con 4 clases

En el ejercicio 2.5 se calcula la matriz de confusión en Python.

Ejercicio 4.12

Ejemplo simple de red neuronal de reconocimiento de patrones para clasificación de datos en diferentes clases (grupos)

Para clasificar datos en clases (grupos), se puede utilizar la herramienta nprtool, y generar código, ver Ejercicio 4.4. Al utilizar esta herramienta se debe tener mucho cuidado y analizar cada línea del código generado.

En este ejercicio se presenta un ejemplo simple sin utilizar la herramienta nprtool. Utilizamos la siguiente función con el conjunto de datos Iris:

```
net1_pattern = patternnet( hiddenSizes, trainFcn, performFcn )
```

Se muestra el código completo:

```
[x1,t1] = iris_dataset;
% Construimos una rede neuronal de una capa oculta de 12 neuronas.
net1_pattern = patternnet(12);
% Entrenamos y mostramos la red
net1_pattern = train(net1_pattern,x1,t1);
view(net1_pattern)
y1 = net1_pattern(x1); % devuelve matriz de 3x150
clases_predecidas = vec2ind(y1); % convertimos a vector de clases
% Performance de la red, por defecto utiliza error cuadrático medio.
performance = perform(net1_pattern,t1,y1)
```

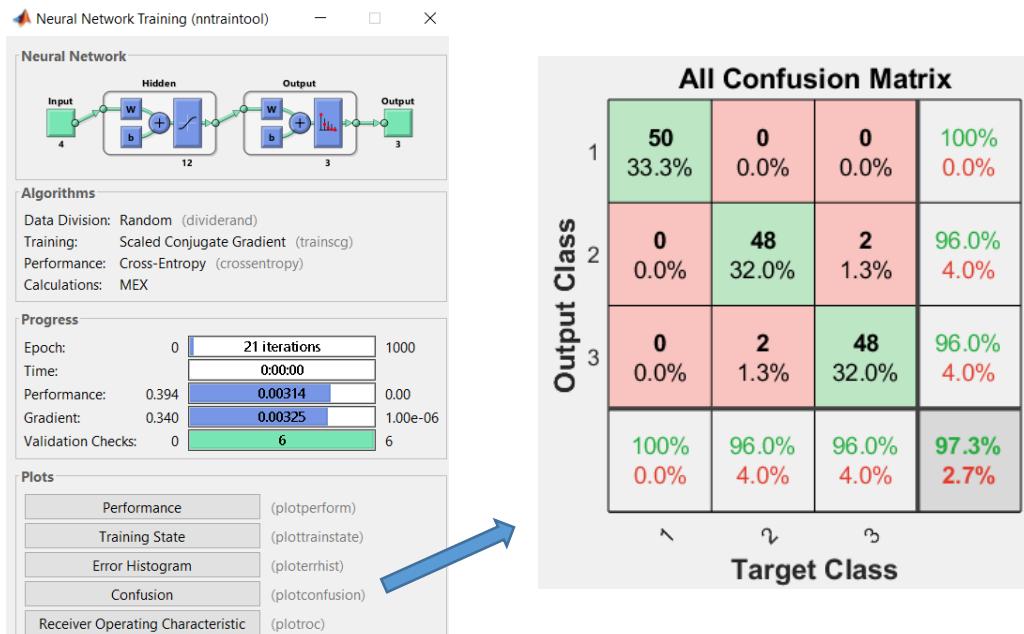


Fig. 71 – Resultados obtenidos del reconocimiento de patrones para conjunto de datos Iris. Izq.) red neuronal y herramienta nntraintool, der.) matriz de confusión con 3 clases



Descarga de los códigos de los ejercicios

Referencias

- Beale, M. H., Hagan, M., & Demuth, H. (2020). Deep Learning Toolbox™ User's Guide. In MathWorks. <https://la.mathworks.com/help/deeplearning/index.html>
- Bishop, C. M. (2006). Bishop, C. M. (2006). Pattern Recognition and Machine Learning. (M. Jordan, J. Kleinberg, & B. Schölkopf, Eds.) Pattern Recognition (Vol. 4, p. 738). Springer. doi:10.1111/1.2819119 Pattern Recognition and Machine Learning. In Pattern Recognition (Vol. 4, Issue 4).
- Del Brío, M. B. y Molina S. (2007). Redes Neuronales y Sistemas Borrosos. 3ra edición. Ed. Alfaomega.
- Demuth H, Beale M, Hagan M. (2018). Neural Network Toolbox™ User's Guide Neural network toolbox. MathWorks.
- Diniz, P. S. R. (2020). Adaptive Filtering. In Adaptive Filtering. Springer International Publishing. <https://doi.org/10.1007/978-3-030-29057-3>
- Diniz, P. S. R. (2013). Adaptive filtering: Algorithms and practical implementation. In Adaptive Filtering: Algorithms and Practical Implementation (Vol. 9781461441069). <https://doi.org/10.1007/978-1-4614-4106-9>
- Goodner, K. L., Dreher, J. G., & Rouseff, R. L. (2001). The dangers of creating false classifications due to noise in electronic nose and similar multivariate analyses. Sensors and Actuators, B: Chemical, 80(3). [https://doi.org/10.1016/S0925-4005\(01\)00917-0](https://doi.org/10.1016/S0925-4005(01)00917-0)
- Gray, R. M., & Davisson, L. D. (2004). An Introduction to Statistical Signal Processing. In An Introduction to Statistical Signal Processing. <https://doi.org/10.1017/cbo9780511801372>
- Papoulis, A., & Hoffman, J. G.. A. Papoulis and S.U. Pillai, Probability, random variables, and stochastic processes, 4th ed., McGraw-Hill, Boston, 2002. In Physics Today (Vol. 20, Issue 1).
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12.
- Pouliarikas, A. D., & Ramadan, Z. M. (2017). Adaptive Filtering Primer With Matlab®. In Adaptive Pouliarikas, A. D., & Ramadan, Z. M. (2019). Wiener filters. In Adaptive Filtering Primer With Matlab®. <https://doi.org/10.1201/9781315221946-4>

- Proakis, J. G., & Manolakis, D. (1998). Digital Signal Processing: Algorithms and applications. Pentice Hall.
- Röck, F., Barsan, N., & Weimar, U. (2008). Electronic nose: Status and future trends. In Chemical Reviews (Vol. 108, Issue 2). <https://doi.org/10.1021/cr068121q>
- Scott, S. M., James, D., & Ali, Z. (2006). Data analysis for electronic nose systems. In Microchimica Acta (Vol. 156, Issues 3–4). <https://doi.org/10.1007/s00604-006-0623-9>
- Su, C., Zhang, Y., Flory, J. H., Weiner, M. G., Kaushal, R., Schenck, E. J., & Wang, F. (2021). Novel clinical subphenotypes in COVID-19: derivation, validation, prediction, temporal patterns, and interaction with social determinants of health.
- Subasi, A. (2020). Data preprocessing. In Practical Machine Learning for Data Analysis Using Python. <https://doi.org/10.1016/b978-0-12-821379-7.00002-3>
- Subasi, A. (2020). Other classification examples. In Practical Machine Learning for Data Analysis Using Python. <https://doi.org/10.1016/b978-0-12-821379-7.00005-9>
- Subasi, A. (2020). Machine learning techniques. In Practical Machine Learning for Data Analysis Using Python. <https://doi.org/10.1016/b978-0-12-821379-7.00003-5>
- Subasi, A. (2020). Classification examples for healthcare. In Practical Machine Learning for Data Analysis Using Python. <https://doi.org/10.1016/b978-0-12-821379-7.00004-7>
- Pytorch, Biblioteca de aprendizaje automático Pytorch, 2021. <https://pytorch.org/>
- Scikit-learn, biblioteca de software de aprendizaje automático para Python, 2021. <https://scikit-learn.org/stable/>
- Tensorflow, Biblioteca de aprendizaje automático Tensorflow, desarrollada por Google, 2021, <https://www.tensorflow.org/>
- The Mathworks, Inc. MATLAB, Version 9.6, 2019. (2019). MATLAB 2019b - MathWorks. In www.Mathworks.Com/Products/Matlab.
- Theodoridis, S., Pikrakis, A., Koutroumbas, K., & Cavouras, D. (2010). Introduction to Pattern Recognition: A Matlab Approach. In Introduction to Pattern Recognition: A Matlab Approach. <https://doi.org/10.1016/C2009-0-18558-6>
- Webb, A. R., & Copsey, K. D. (2011). Statistical Pattern Recognition: Third Edition. In Statistical Pattern Recognition: Third Edition. <https://doi.org/10.1002/9781119952954>
- Yu, H., Wang, J., Zhang, H., Yu, Y., & Yao, C. (2008). Identification of green tea grade using different feature of response signal from E-nose sensors. Sensors and Actuators, B: Chemical, 128(2). <https://doi.org/10.1016/j.snb.2007.07.048>

Capítulo V - Redes Neuronales con Mapas Autoorganizados (SOM)

Las redes neuronales autoorganizadas, conocidas como SOM, son técnicas complejas no supervisadas. Proyectan los datos a un nuevo espacio generando mapas con representaciones discretas. Las neuronas que componen la red se autoorganizan y compiten entre sí. Utilizan funciones de proximidad o vecindad. Son muy útiles para reducir la dimensión de los datos de entrada, los resultados generalmente se muestran en mapas de dos dimensiones. Las SOMs también se conocen como mapas de Kohonen debido a que el modelo fue descrito por primera vez por el profesor Teuvo Kohonen. Se presentan ejemplos en Python y en Matlab

Introducción

Una clase particularmente interesante de sistema no supervisado se basa en el aprendizaje competitivo, en el que las neuronas de salida compiten entre sí para activarse, donde solo se activa las neuronas ganadoras. Tal competencia se puede implementar al tener conexiones con caminos de retroalimentación entre las neuronas. Las neuronas se ven obligadas a organizarse. Dicha red se denomina Mapa Autoorganizado (en inglés, Self Organizing Map, SOM).

El objetivo principal de una SOM es transformar un patrón de una señal de entrada de dimensión arbitraria en un nuevo mapa discreto de una o dos dimensiones, se debe realizar esta transformación en forma adaptativa y topológicamente ordenada. Por lo tanto, configuramos nuestra SOM colocando neuronas en los nodos de una red unidimensional o bidimensional. Se pueden usar mapas de mayor dimensión, pero generalmente no se usan (The Mathworks, 2019).

Las neuronas se sintonizan selectivamente con varios patrones de entrada (estímulos) o clases de patrones de entrada durante el proceso del aprendizaje competitivo. Las ubicaciones de las neuronas se ordenan y se crea un sistema de coordenadas nuevo para las entradas de la red. Podemos ver esto como una generalización no lineal del PCA (análisis de componentes principales).

En la Fig. 72 se muestra la proyección de datos en un mapa autoorganizado. Las entradas x del espacio de entrada se mapean al espacio de salida obteniendo los puntos $I(x)$. Cada punto I en el espacio de salida se asigna a un punto correspondiente $w(I)$ en el espacio de entrada.

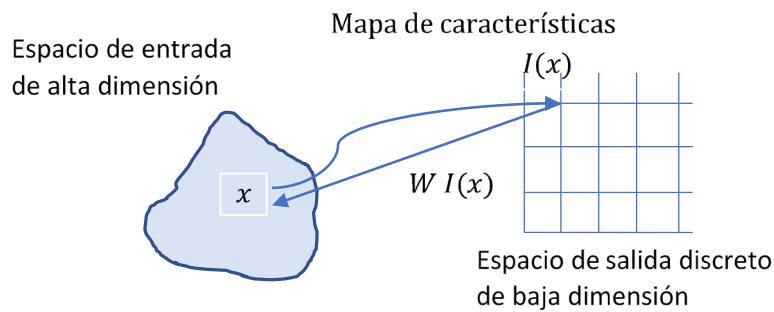


Fig. 72 – Proyección de datos en el mapa autoorganizado

Analizaremos el tipo particular de SOM conocido como Red Kohonen. Esta red SOM contiene una estructura del tipo feedforward con una sola capa computacional dispuesta en filas y columnas. Cada neurona está completamente conectada a todos los nodos de origen en la capa de entrada. En la Fig. 73 se muestra la proyección de datos para estas redes neuronales. Si el mapa fuera unidimensional solo tendrá una fila (o una sola columna) en la capa computacional.

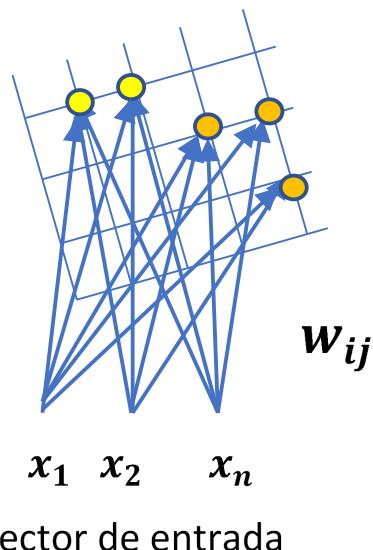


Fig. 73 – Proyección de datos en el mapa autoorganizado Kohonen

Componentes de la autoorganización

El proceso de autoorganización involucra cuatro componentes principales (Beale, 2020):

- Inicialización: se inicializan todos los pesos colocando valores pequeños aleatorios.
- Competencia: en cada patrón de entradas, las neuronas de la red calculan los valores de una función discriminante que proporciona la base para la competencia. Donde la neurona particular con el valor más pequeño de la función discriminante se considera ganadora.
- Cooperación: se toma la neurona ganadora para determinar la ubicación espacial de la vecindad topológica que tiene neuronas excitadas, proporcionando así la cooperación entre neuronas vecinas (próximas).

- Adaptación: Las neuronas excitadas reducen sus valores de función discriminante con respecto al patrón de entrada mediante un ajuste adecuado de los pesos de conexión asociados. De esta forma se mejora la respuesta de la neurona ganadora a la aplicación posterior de un patrón de entrada similar.

En el proceso de competencia podemos definir nuestra función discriminante como la distancia euclíadiana entre el vector de entrada x y el vector de peso w_j elevada al cuadrado para cada neurona j , ecuación (5.1).

$$d_j(x) = \sum_{i=1}^D (x_i - w_{ji})^2 \quad (5.1)$$

Es decir, la neurona con el vector de peso que se acerca más al vector de entrada (la que es más similar a él) se la declara ganadora. De esta forma, el espacio de entrada continuo se puede relacionar con el espacio de salida discreto de las neuronas mediante un proceso simple de competencia entre las neuronas (Demuth, 2018), (Pytorch, 2021), (Tensorflow, 2021).

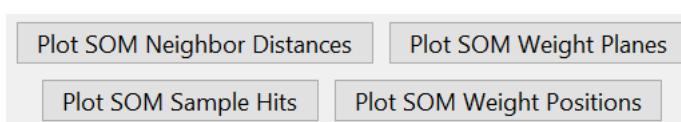
Ejercicios

Ejercicio 5.1

Redes Neuronales SOM con Matlab

Agrupamiento de datos con redes neuronales SOM

- Abrir la herramienta de Agrupamientos de datos de Matlab® (nctool), al cargar datos, seleccionar “simplecluster_dataset”. Se pueden cargar datos de ejemplo de Matlab® o datos propios. Ingresar 10 neuronas para la capa oculta. Luego, pulsar el comando para “entrenar” la red neuronal. Presionar los siguientes comandos para graficar los resultados (The Mathworks, 2019).



En la Fig. 74 se muestra la topología de la red, y algunos resultados. En la pantalla final se puede generar el código de programa (script) mediante el comando “Advanced Script”.

- Modificar el tamaño del mapa 2D y analizar los cambios generados en los resultados.

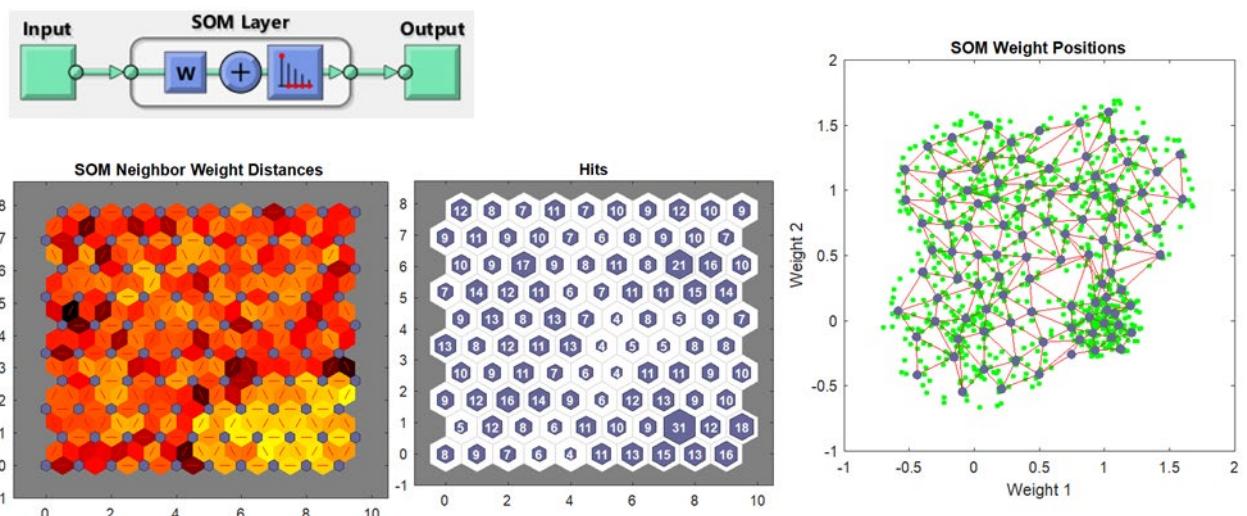


Fig. 74 – Resultados del Agrupamiento de Datos con redes SOM en Matlab

A continuación, se muestra el programa modificado para una mejor comprensión, a su vez se modifican las primeras líneas para importar los datos correspondientes. Se dispone de un script independiente de la herramienta nctool.

Notar que solo se utilizan los datos de entradas, no se utilizan las categorías deseadas, es un aprendizaje No supervisado.

```
% Ejemplo de Agrupamiento con redes neuronales SOM (Self-Organizing Map)
% Cargamos los datos x: entradas, no se utilizan las categorías deseadas (No supervisado)
clear all; close all; clc
load simplecluster_dataset
x = simpleclusterInputs; % Solo utiliza entradas. No se utilizan las categorías deseadas
% Creamos Mapa Autoorganizado (SOM)
dimension1 = 10;
dimension2 = 10;
net1 = selforgmap([dimension1 dimension2]);
% Elegimos las funciones para graficar, ver ayudas en help nnplot
net1.plotFcns = {'plotsomtop','plotsomnc','plotsomnd',...
    'plotsomplanes', 'plotsomhits', 'plotsompos'};
% Entrenamos la red:
[ net1, tr ] = train( net1, x );
% Testeamos y mostramos la red
y = net1(x);
view(net1)
% Graficamos, agregar comentarios si no se quieren mostrar todos los gráficos
figure, plotsomtop(net1);
figure, plotsomnc(net1) ;
figure, plotsomnd(net1) ;
figure, plotsomplanes(net1);
figure, plotsomhits(net1,x);
figure, plotsompos(net1,x);
```

```
% Desarrollo: Generamos código de programa con función de red neuronal
genFunction(net1,'NeuralNetworkFunction_1');
y1 = NeuralNetworkFunction_1(x);
% Generamos función de red neuronal para matrices de entrada (no array de celdas)
genFunction(net1,'NeuralNetworkFunction_2','MatrixOnly','yes');
y1 = NeuralNetworkFunction_2(x);
% Generamos diagrama Simulink
gensim(net1);
```

Ejercicio 5.2**Redes Neuronales SOM con Python**

En este ejemplo se agrupan datos correspondientes a 5 clases. Se utiliza la librería sklearn_som.som. Se obtiene una precisión del 97,18 %. En la Fig. 75 se comparan los resultados de las clases originales y las clases predecidas.

```
import matplotlib.pyplot as plt # ##
from matplotlib.colors import ListedColormap
from sklearn_som.som import SOM
from sklearn.datasets import make_blobs
# Armamos el conjunto de datos y lo graficamos
nro_muestras = 320
centros_1 = ( [-2, -2], [-2, 2], [1.5, -2.2],[1.6, 3],[3, 0])
datos_1, labels = make_blobs(n_samples=nro_muestras,
                             centers=centros_1, cluster_std=0.6, random_state = 0 ) #
colores = ( 'red', 'blue', 'green', 'yellow', 'magenta' )
marcadores = [ 'o', (5,1), ' ', '^', 'v', '<', '>', 's', 'd', '!', 'x', '+' ]
plt.rc('axes', titlesize=18) ; plt.rc('font', size=16) ;
fig, ax1 = plt.subplots( figsize = (8,4) )
for n_clases1 in range(len(centros_1)):
    ax1.scatter(datos_1[labels==n_clases1][:, 0], datos_1[labels==n_clases1][:, 1],
                c = colores[n_clases1], s=60, label=str(n_clases1) ,marker = marcadores[n_clases1] )
ax1.grid() ; ax1.legend(loc="upper right") ;
X1 = datos_1 ; y1 = labels
# Creamos una red a 5x1 SOM (de 5 clústers)
som = SOM(m=5, n=1, dim=2, random_state=1234)
# Ajustamos la red con los datos y predecimos los datos
som.fit(X1)
y_predec = som.predict(X1)
# Graficamos
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,6))
x = X1[:,0] ; y = X1[:,1]
colors= ( 'red', 'blue', 'green', 'yellow', 'magenta')
ax[0].scatter(x, y, c=y1, cmap=ListedColormap(colors))
ax[0].title.set_text('Clases verdaderas')
colors = ( 'green', 'red', 'yellow','blue', 'magenta')
ax[1].scatter(x, y, c=y_predec, cmap=ListedColormap(colors))
ax[1].title.set_text('Clases predecidas con red neuronal SOM')
```

```

plt.savefig('Resultado1.png')
# Ajustamos las etiquetas y calculamos precisión
y1[y1==0]=40 ; y1[y1==1]=30 ; y1[y1==2]=0; y1[y1==3]=20; y1[y1==4]=10
y_predec[y_predec==1]=10 ; y_predec[y_predec==2]=20 ; y_predec[y_predec==3]=30
y_predec[y_predec==4]=40 ;
from sklearn.metrics import accuracy_score
acc= accuracy_score(y1, y_predec)
print('Acc: {:.4f}'.format(acc))
from sklearn.metrics import f1_score
print ("f1 score macro: ", f1_score(y1, y_predec, average='macro') )

```

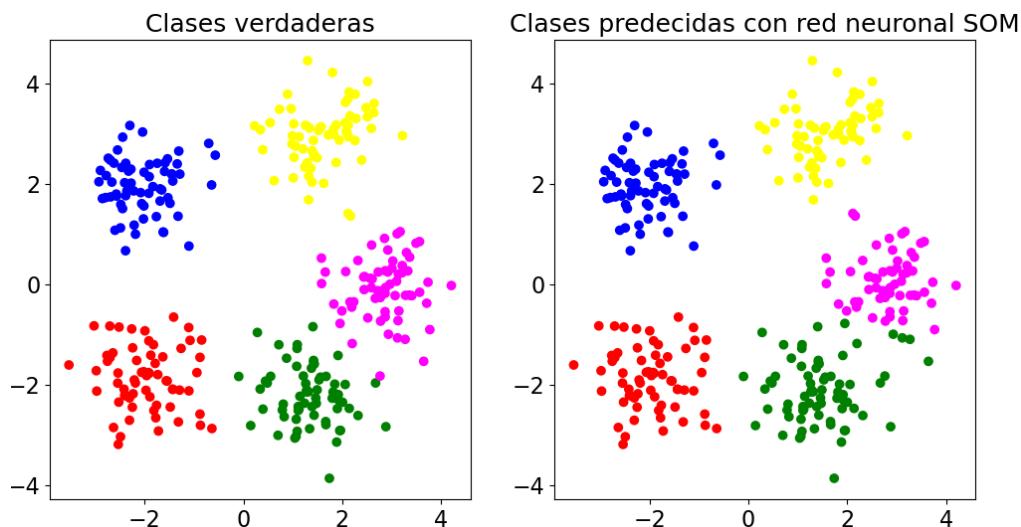


Fig. 75 – Resultados del Agrupamiento de Datos con redes SOM con módulo sklearn

Ejercicio 5.3

Redes Neuronales SOM con librería sompy en Python

En este ejemplo se utilizan las librerías sompy (Moosavi, 2014). Se generan 2 conjuntos de datos y se muestran los mapas SOM obtenidos, ver Fig. 76 y Fig. 77.

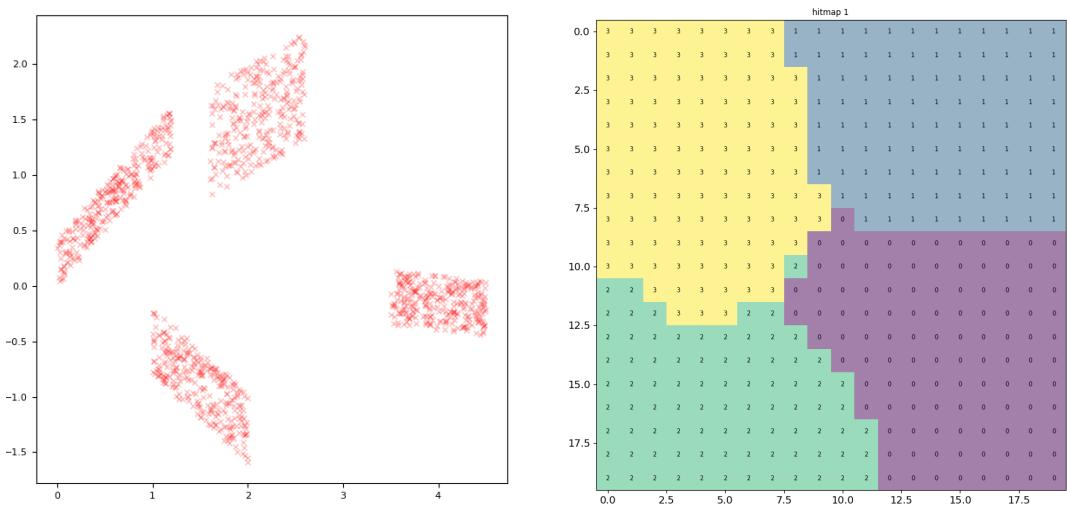


Fig. 76 – Izq.) Conjunto de datos, der.) mapa SOM obtenido

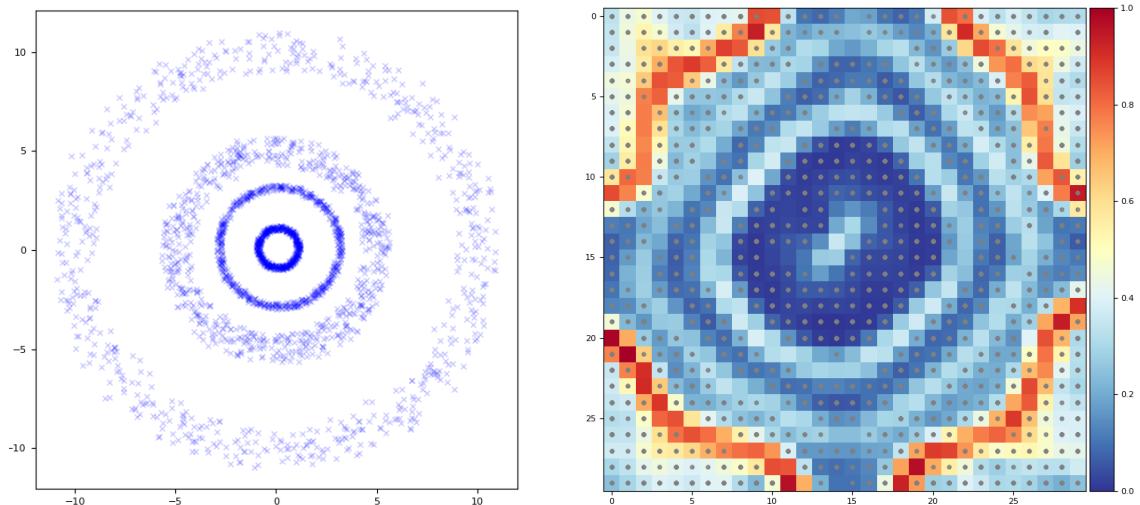


Fig. 77 – Izq.) Conjunto de datos circulares, der) mapa SOM obtenido



Descarga de los códigos de los ejercicios

Referencias

- Beale, M. H., Hagan, M., & Demuth, H. (2020). Deep Learning Toolbox™ User's Guide. In MathWorks. <https://la.mathworks.com/help/deeplearning/index.html>
- Del Brío, M. B. y Molina S. (2007). Redes Neuronales y Sistemas Borrosos. 3ra edición. Ed. Alfaomega.
- Demuth H, Beale M, Hagan M. (2018). Neural Network Toolbox™ User's Guide Neural network toolbox. MathWorks.
- Moosavi, V, Packmann, S and Vall. (2014) SOMPY: A Python Library for Self Organizing Map (SOM)}
[https://github.com/sevamoo/SOMPY}](https://github.com/sevamoo/SOMPY).
- Pytorch, Biblioteca de aprendizaje automático Pytorch, (2021). <https://pytorch.org/>
- The Mathworks, Inc. MATLAB, Version 9.6, 2019. (2019). MATLAB 2019b - MathWorks. In www.Mathworks.Com/Products/Matlab.

Capítulo VI - Redes Neuronales Dinámicas

En este capítulo se presentan diferentes redes neuronales dinámicas. Estas redes son muy adecuadas cuando se trabaja con entradas secuenciales, donde importa el orden de los datos. Contienen memoria y bloques de retardo. Se presentan las estructuras y los algoritmos más utilizados. En los ejercicios se analiza y se compara la respuesta de redes neuronales dinámicas y estáticas. También se muestran muchos ejemplos avanzados de estas redes dinámicas modernas que resultan muy potentes

Introducción

Las redes neuronales pueden ser estáticas o dinámicas. En las redes estáticas la salida se calcula directamente a partir de la entrada mediante conexiones feedforward. Mientras que, en las redes dinámicas, la salida depende no solo de la entrada presente o actual de la red, sino también de entradas y salidas o estados que sean anteriores o actuales de la red. A modo de ejemplo, las redes con filtros adaptativos son redes dinámicas, debido a que la salida se calcula a partir líneas de retardo correspondientes a entradas anteriores. Las redes Hopfield también son dinámicas, debido a que tienen conexiones recurrentes (de retroalimentación), por lo tanto, la salida actual depende tiempo presente y de tiempos anteriores. Es decir que las redes dinámicas tienen memoria, su respuesta en un momento determinado dependerá de la entrada actual, y del historial de las entradas (Hagan, 2014).

Las redes neuronales se entranan con algoritmos que se basan en gradientes tales como el descenso más pronunciado y algoritmo de gradiente conjugado. También se pueden entrenar con Jacobianos, como los algoritmos de Gauss-Newton y Levenberg-Marquardt. El entrenamiento de redes estáticas y dinámicas difiere en la manera en que se calcula el Jacobiano o la matriz de gradiente.

Las redes dinámicas contienen bloques de retardos, que trabajan entradas secuenciales. Es importante el orden que ingresan las entradas. Las redes dinámicas pueden tener solamente conexiones feedforward, como por ejemplo los filtros adaptativos, o también pueden contener algunas conexiones recurrentes de retroalimentación, llamadas redes neuronales recurrentes (RNN).

Debido a que las redes dinámicas tienen memoria, se pueden entranar para aprender patrones secuenciales o variables en el tiempo. En lugar de aproximar funciones, como la red estática de perceptrones multicapa, una red dinámica puede aproximarse a un sistema dinámico. Esto tiene aplicaciones en áreas tan diversas como control de sistemas dinámicos, predicción en mercados financieros, equalización de canales en sistemas de comunicación, clasificación, detección de fallas, reconocimiento de voz, etc.

Las redes dinámicas se pueden entranar usando métodos de optimización estándar de redes estáticas. Sin embargo, los gradientes y jacobianos que se requieren para estos métodos no se pueden calcular usando el algoritmo de propagación hacia atrás estándar. Se utilizan algoritmos de propagación hacia atrás dinámicos para calcular los gradientes de las redes dinámicas. Hay dos enfoques generales (con muchas variaciones) para el gradiente y los cálculos

jacobianos en redes dinámicas: retropropagación en el tiempo (BPTT) y aprendizaje recurrente en tiempo real (RTRL).

En el algoritmo BPTT, la respuesta de la red se calcula para todos los puntos de tiempo, y luego el gradiente se calcula comenzando en el último punto de tiempo y trabajando hacia atrás en el tiempo. Este algoritmo es eficiente para el cálculo de gradientes, pero es difícil de implementar en línea, porque el algoritmo trabaja hacia atrás en el tiempo desde el último paso de tiempo.

En el algoritmo RTRL, su gradiente se puede estimar al mismo tiempo que la respuesta de la red, ya que se calcula comenzando en el primer punto de tiempo y luego avanzando en el transcurso tiempo. RTRL requiere más cálculos que BPTT para calcular el gradiente, pero RTRL permite un marco conveniente para la implementación en línea. Para los cálculos jacobianos, el algoritmo RTRL es generalmente más eficiente que el algoritmo BPTT (Pytorch, 2021), (Beale, 2020).

Estructuras de Redes estáticas, dinámicas prealimentadas y dinámicas recurrentes

Las redes neuronales pueden tener conexiones solamente prealimentadas (también llamadas unidireccionales, de alimentación directa, o en inglés feedforward) o pueden tener conexiones prealimentadas y conexiones de retroalimentación o recurrentes (The Mathworks, 2019).

Las redes estáticas solo tienen conexiones prealimentadas (feedforward).

Existen 2 clases de redes dinámicas:

- Solo con conexiones prealimentadas
- Con conexiones prealimentadas y conexiones de retroalimentación o recurrentes (RNN)

Aprendizaje en redes dinámicas

Consideremos el ejemplo simple de la Fig. 89, con el método de gradiente descendente (Demuth, 2018). El primer paso es calcular el gradiente de la función de rendimiento. Para este ejemplo usaremos la suma errores al cuadrado:

$$F(x) = \sum_{t=1}^Q e^2(t) = \sum_{t=1}^Q (t(t) - a(t))^2 \quad (6.1)$$

Calculamos las derivadas parciales:

$$\frac{\partial F(x)}{\partial lw_{1,1}(1)} = \sum_{t=1}^Q \frac{\partial e^2(t)}{\partial lw_{1,1}(1)} = -2 \cdot \sum_{t=1}^Q e(t) \cdot \frac{\partial a(t)}{\partial lw_{1,1}(1)} \quad (6.2)$$

$$\frac{\partial F(x)}{\partial iw_{1,1}} = \sum_{t=1}^Q \frac{\partial e^2(t)}{\partial iw_{1,1}} = -2 \cdot \sum_{t=1}^Q e(t) \cdot \frac{\partial a(t)}{\partial iw_{1,1}} \quad (6.3)$$

Para una red estática, estos términos son fáciles de calcular $\frac{\partial a(t)}{\partial lw_{1,1}(1)}$; $\frac{\partial a(t)}{\partial iw_{1,1}}$ corresponderían a $a(t-1)$ y $p(t)$, respectivamente. Sin embargo, para redes recurrentes, los pesos tienen dos efectos en la salida de la red. El primero es el efecto directo, que también se ve en la red estática correspondiente. El segundo es un efecto indirecto, causado por el hecho de que una de las entradas de la red es una salida de la red anterior. Calculemos las derivadas de la salida de la red para analizar estos dos efectos.

$$a(t) = iw_{1,1} \cdot p(t) + lw_{1,1}(1) \cdot a(t-1) \quad (6.4)$$

$$\frac{\partial a(t)}{\partial lw_{1,1}(1)} = \frac{\partial}{\partial lw_{1,1}(1)} (iw_{1,1} \cdot p(t)) + a(t-1) \quad (6.5)$$

Obtenemos:

$$\frac{\partial a(t)}{\partial lw_{1,1}(1)} = lw_{1,1}(1) \cdot \frac{\partial a(t-1)}{\partial lw_{1,1}(1)} + a(t-1) \quad (6.6)$$

$$\frac{\partial a(t)}{\partial iw_{1,1}} = p(t) + lw_{1,1}(1) \cdot \frac{\partial a(t-1)}{\partial iw_{1,1}} \quad (6.7)$$

Los términos $a(t-1)$ y $p(t)$ representan el efecto directo que cada peso aporta a la salida de la red.

Los términos con derivadas representan el efecto indirecto. A diferencia del cálculo del gradiente para redes estáticas, la derivada en cada punto de tiempo depende de la derivada en puntos de tiempo anteriores (en otros casos podría depender de puntos de tiempo futuros).

Analizamos una red neuronal de una sola neurona. Ahora analizaremos una red multicapa estática con un solo bucle de retroalimentación agregado desde la salida de la red a la entrada a través de un solo retardo. En la Fig. 78 el vector x representa todos los parámetros de la red (pesos y sesgos) y el vector $a(t)$ representa la salida de la red multicapa en el paso de tiempo t .

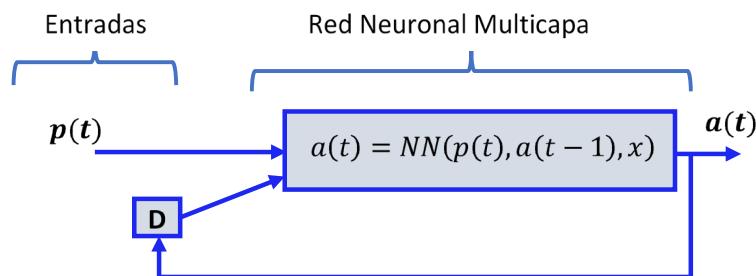


Fig. 78 –Red dinámica recurrente simple

Queremos ajustar los pesos y los sesgos de la red para minimizar el índice de rendimiento $F(x)$, que normalmente se elige el error cuadrático medio. Derivamos $F(x)$ para minimizar el error. Para redes dinámicas, se modifica el algoritmo de propagación hacia atrás estándar de redes estáticas. Existen dos enfoques diferentes para este problema. Ambos usan la regla de derivación de la cadena, pero se implementan de diferentes maneras según 2 métodos distintos. El primer método corresponde al aprendizaje recurrente en tiempo real (RTRL)

$$\frac{\partial F}{\partial x} = \sum_{t=1}^Q \left[\frac{\partial a(t)}{\partial x^T} \right]^T x \frac{\partial^e F}{\partial a(t)} \quad (6.8)$$

El superíndice e indica derivada explícita, sin tener en cuenta los efectos indirectos a lo largo del tiempo. Las derivadas explícitas se pueden obtener con el algoritmo de propagación hacia atrás básico visto anteriormente. Para encontrar las derivadas completas necesitamos la ecuación adicional, donde el término $\frac{\partial a(t)}{\partial x^T}$ que debe propagarse en el tiempo

$$\frac{\partial a(t)}{\partial x^T} = \frac{\partial^e a(t)}{\partial x^T} + \frac{\partial^e a(t)}{\partial a^T(t-1)} \cdot \frac{\partial a(t-1)}{\partial x^T} \quad (6.9)$$

El segundo método corresponde al de aprendizaje con propagación hacia atrás a través del tiempo (BPTT).

$$\frac{\partial F}{\partial x} = \sum_{t=1}^Q \left[\frac{\partial^e a(t)}{\partial x^T} \right]^T \cdot \frac{\partial F}{\partial a(t)} \quad (6.10)$$

Donde necesitamos la ecuación adicional $\frac{\partial F}{\partial a(t)}$ que debe propagarse hacia atrás en el tiempo.

$$\frac{\partial F}{\partial a(t)} = \frac{\partial^e F}{\partial a(t)} + \frac{\partial^e a(t+1)}{\partial a^T(t)} x \frac{\partial F}{\partial a(t+1)} \quad (6.11)$$

El algoritmo RTRL requiere más cálculo que el algoritmo BPTT para calcular el gradiente. Sin embargo, el algoritmo BPTT no se puede implementar convenientemente en tiempo real, ya que las salidas deben calcularse para todos los pasos de tiempo, y luego las derivadas deben propagarse hacia atrás al punto de tiempo inicial. El algoritmo RTRL es muy adecuado para la implementación en tiempo real, ya que las derivadas se pueden calcular en cada paso de tiempo. Para los cálculos jacobianos, que son necesarios para los algoritmos de Levenberg-Marquardt, el algoritmo RTRL suele ser más eficiente que el algoritmo BPTT.

Las redes dinámicas se pueden entrenar usando los mismos procedimientos de optimización para redes estáticas multicapa. Sin embargo, el cálculo del gradiente para redes dinámicas es más complejo que para redes estáticas. El enfoque de propagación hacia atrás a través del tiempo (BPTT), comienza en el último punto de tiempo y trabaja hacia atrás en el tiempo para calcular el gradiente. El segundo enfoque es el aprendizaje recurrente en tiempo

real (RTRL), comienza en el primer momento y luego avanza en el tiempo. RTRL generalmente requiere menos almacenamiento que BPTT.

Predicción e Identificación con redes neuronales dinámicas para aplicaciones de series temporales no lineales.

La predicción es un filtrado dinámico, en el que se utilizan valores pasados de una o más series de tiempo para predecir valores futuros. Mediante las redes neuronales dinámicas, con líneas de retardo, se realiza el filtrado y la predicción no lineal.

Hay muchas aplicaciones en las que se usan la predicción, por ejemplo, análisis financiero para predecir el valor futuro de una acción o un bono, o un mecánico podría querer predecir la posible falla o mal funcionamiento de un motor.

Los modelos predictivos también son utilizados para la identificación de sistemas (o modelado dinámico), en los que se crean modelos dinámicos de sistemas para conocer su funcionamiento.

Redes neuronales recurrentes (RNN)

Los modelos de retroalimentación, como las redes neuronales de perceptrón multicapa (MLP) y las redes convolucionales (CNN), utilizan datos de entrada de tamaño fijo (vectores de dimensionalidad fija) y asignan etiquetas de salida o valores analógicos. Son muy poderosos y se utilizan con éxito para muchas tareas. Ambos demostraron ser adecuados para resolver problemas de predicción de regresión y de clasificación; las CNN se han convertido en el método de referencia para problemas de predicción que involucren imágenes. Sin embargo, muchos datos no están en forma de vectores de tamaño fijo, sino que existen en forma de secuencias. Las RNN fueron diseñados para poder manejar datos secuenciales. A través de un tipo de memoria interna, la red está capacitada para retener importantes entradas previas, que alimentan las predicciones futuras (Maas, 2011).

Las RNN tienen conexiones recurrentes (de retroalimentación), lo que significa que se calcula la salida actual en función de las salidas de tiempo presente y tiempos pasados.

Una RNN simple se puede explicar como una colección de redes neuronales organizadas una tras otra, cada una de las cuales transmite un mensaje a otra. Estas redes tienen una memoria que almacena el conocimiento sobre los datos vistos, pero su memoria es a corto plazo y no puede mantener series de tiempo a largo plazo (Pytorch, 2021).

La Fig. 79, a la izquierda muestra una RNN, a la derecha muestra la red detallada con su evolución temporal.

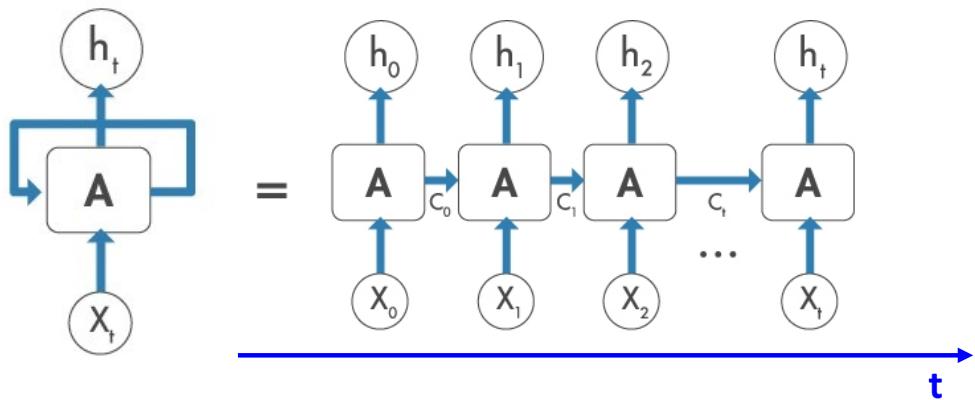


Fig. 79 – Esquema de Red Neuronal Recurrente (RNN)

Las arquitecturas van desde redes completamente interconectadas (Fig. 80) hasta redes que están parcialmente conectadas (Fig. 81), incluidas redes de alimentación de múltiples capas con distintas capas de entrada y salida. Las redes completamente conectadas no tienen capas de entrada distintas de nodos, y cada nodo tiene entrada de todos los demás nodos. Es posible la retroalimentación al propio nodo.

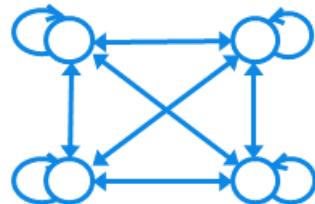


Fig. 80 – Ejemplo de una red neuronal recurrente (RNN) totalmente conectada.

En la Fig. 81 se utiliza red neuronal simple parcialmente recurrentes. Aunque algunos nodos son parte de una estructura de retroalimentación, otros nodos proporcionan el contexto secuencial y reciben retroalimentación de otros nodos. Los pesos de las unidades de contexto (C_1 y C_2) se procesan como los de las unidades de entrada, por ejemplo, usando propagación hacia atrás. Las unidades reciben retroalimentación retardada desde las unidades de la segunda capa.

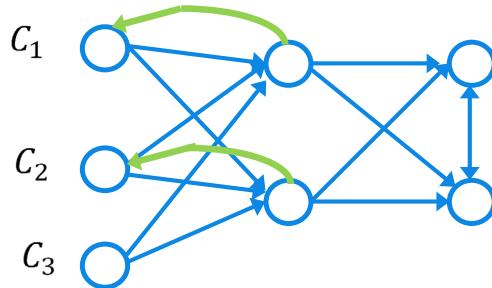


Fig. 81 – Ejemplo de una red recurrente parcialmente conectada

Ventajas de las RNN. Procesamiento natural de lenguaje (PNL)

Si tomamos como ejemplo una RNN para el pronóstico del tiempo, se puede pronosticar si va a llover o granizar utilizando la información de una secuencia previa. Esta secuencia de datos puede corresponder a diferentes datos, imágenes y videos. Después de cada información nueva, actualizamos lentamente la probabilidad de lluvia o granizo y finalmente llegamos a una conclusión. Es decir que las RNN tienen memoria (Scikit-learn, 2021), (Tensorflow, 2021).

Estas redes y específicamente una variante de RNN, la red Long Short Term Memory (LSTM), posiblemente han obtenido el mayor éxito al trabajar con secuencias de palabras y párrafos, generalmente denominado procesamiento del lenguaje natural (PNL), tanto en modelos predictivos como generativos.

Muchas publicaciones muestran que las redes neuronales artificiales recurrentes (RNN) son eficaces para procesar lenguaje humano natural (PNL). Las RNN se utilizan mucho en tareas de PNL como generación de texto, traducción automática, subtítulos de imágenes, reconocimiento de sentimientos, etc. En las tareas de PNL, se utilizan dos pasos estandarizados: i) procesar el texto en forma de vectores y ii) entrenar una red neuronal mediante estos vectores. La red neuronal artificial seleccionada puede ser RNN, CNN o incluso una red neuronal de retroalimentación para completar esta tarea. Sin embargo, se utilizan mucho las RNN con datos secuenciales, ya que los datos de texto son de naturaleza secuencial. Justamente las RNN son redes artificiales donde sus conexiones entre neuronas forman secuencias temporales.

Las RNN son capaces de obtener o capturar información temporal dinámica (utiliza memoria temporal). Estas redes provienen de las redes neuronales del tipo feedforward, pero ofrecen muchas ventajas.

Existen tres limitaciones principales de las redes de alimentación directa (feedforward) por las que estas redes muchas veces resultan inadecuadas para los datos de secuenciales:

- No tiene en cuenta el orden.
- Utiliza un tamaño de entrada fijo.
- No puede generar predicciones para diferentes longitudes.

Una de las características fundamentales de los datos de texto es la importancia que tiene su orden. Reorganizar o cambiar el orden de las palabras en una frase u oración puede cambiar o distorsionar su significado por completo. Aquí es donde surge la limitación de la red neuronal feedforward. En una red neuronal feedforward, el orden de los datos no se puede tener en cuenta debido a esta limitación.

Clasificación de RNN

Las RNN se pueden clasificar según sus conexiones, ver Fig. 82. En la figura cada rectángulo corresponde a un vector, mientras que las flechas representan diferentes funciones (por ejemplo, multiplicación de matrices). Los vectores de entrada se representan con rectángulos rojos, los vectores de salida están en verde y los rectángulos en azul contienen el estado de la RNN. Se describen de izquierda a derecha:

- Una a una: modo de procesamiento vainilla sin RNN, con entrada de tamaño fijo y salida de tamaño fijo, como, por ejemplo, clasificación de imágenes.

- Una a muchas: salida de secuencia (por ejemplo, los subtítulos de la imagen, se toma una imagen y se genera una oración de palabras).
- Muchas a una: entrada de secuencia (por ej., análisis de sentimientos en el que una oración determinada se clasifica como que expresa un sentimiento positivo o negativo o si se le da un texto para predecir el siguiente carácter).
- Muchas a muchas: entrada de secuencia y salida de secuencia. Por ejemplo, traducción automática: una RNN lee una oración en inglés y la traduce.
- Muchas a muchas: entrada y salida para secuencias sincronizadas. Por ejemplo, clasificación de video donde se etiqueta cada uno de los fotogramas del video.

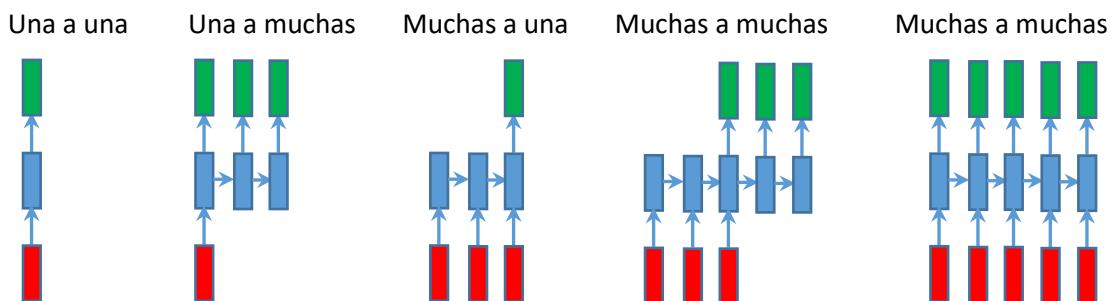


Fig. 82 –Clasificación de RNN según su conexión

Redes recurrentes LSTM y GRU

Las RNN utilizan información previa guardada en la memoria, se guarda como un "estado" adentro de la RNN (Tensorflow, 2021). Existen muchas variantes de RNN, las más utilizadas son:

- RNN simple (simple, conocida como vainilla).
- Redes de memoria a corto plazo (LSTM).
- Redes de unidades recurrentes cerradas (GRU).

Para combatir el problema de memoria a largo plazo, se crearon dos RNN especializadas que hacen uso de un mecanismo interno llamado "compuertas" que pueden regular el flujo de la información. Esta técnica ayuda a la red a aprender qué datos en una secuencia son importantes mantener (agregándolos al estado oculto) y cuáles deben desecharse. Al hacerlo, hace que la red sea más capaz de aprender las dependencias a largo plazo. Las 2 redes más populares en la actualidad se denominan redes con Memoria a corto plazo largo (LSTM) y redes con Unidades recurrentes cerradas (GRU). La mayoría de los resultados de vanguardia científicos y tecnológicos basados en RNN se logran con estas dos redes. Las LSTM y GRU se utilizan en el reconocimiento de voz, en la síntesis de voz, la generación de texto y la generación de subtítulos para videos.

Ni los LSTM ni los GRU tienen arquitecturas fundamentalmente diferentes de las RNN simples. El flujo de control sigue siendo similar: procesa datos transmitiendo información a medida que se va propagando. Las diferencias radican en cómo se calcula su estado oculto y las operaciones realizadas dentro de las redes o celdas LSTM o GRU.

Las redes GRU utilizan dos puertas: actualizar y restablecer. Mientras que las redes LSTM se agregan 2: olvidar y salida. Las redes GRU pasan directamente el estado oculto a la siguiente unidad, mientras que las redes LSTM utilizan la celda de memoria para calcular el estado oculto.

El rendimiento de LSTM resulta mejor para conjunto de datos grandes. Pero para pocos parámetros las redes GRU convergen más rápido.

En la Tabla VIII se describen las compuertas y sus funciones. En la Fig. 83 se muestra un gráfico con las dependencias para GRU, y en la Fig. 84 las dependencias para LSTM. Siendo $x^{(t)}$ la entrada, $a^{(t)}$ y $c^{(t)}$ la salida de la red y el estado.

Tabla VIII – Tipo de compuertas para redes GRU y LSTM

Tipo de compuerta (gate)	Función	Uso
Compuerta de actualización (update gate): Γ_u	Determina el largo de los datos pasados	GRU, LSTM
Compuerta de restablecer o relevancia (relevancia gate): Γ_r	Determina si eliminar la información anterior	GRU, LSTM
Compuerta de olvido (forget gate): Γ_f	Determina si borrar o no una celda	LSTM
Compuerta de salida (output gate): Γ_o	Determina cuanta información dejar pasar a la salida	LSTM

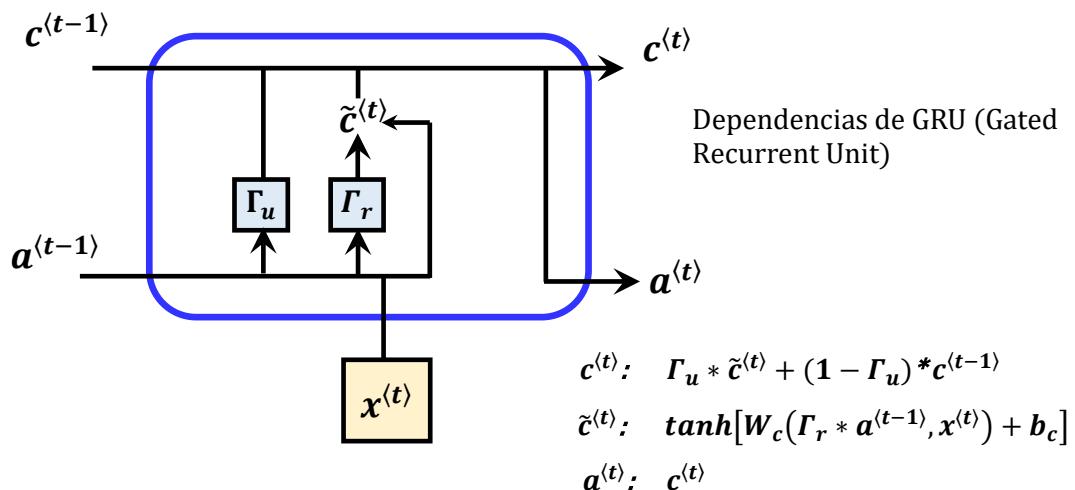


Fig. 83 – Diagrama de dependencias para GRU

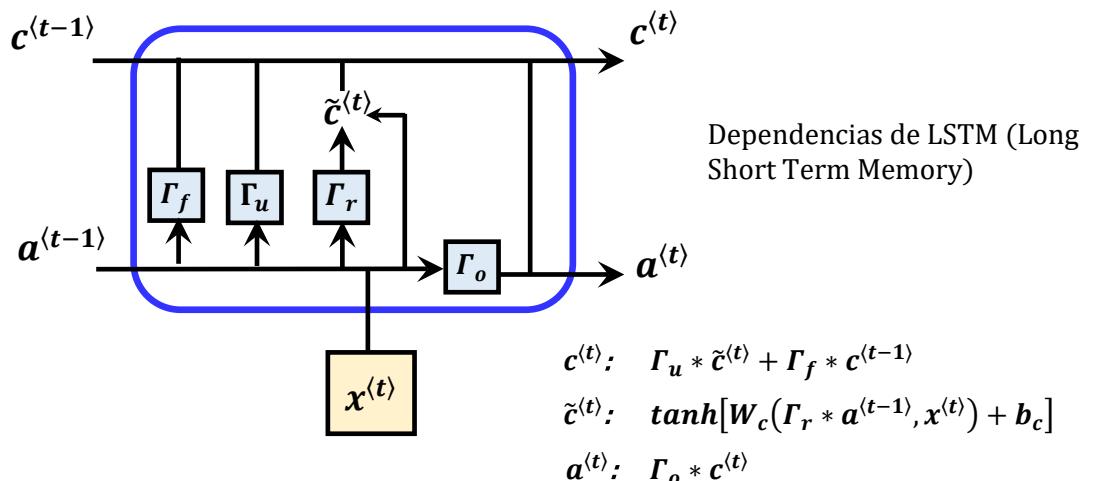


Fig. 84 – Diagrama de dependencias para LSTM

Ejercicios de Redes neuronales dinámicas

Ejercicio 6.1

Análisis de Respuesta de Red Neuronales dinámicas y estática

Se pide armar una red neuronal estática, una dinámica no recurrente y otra dinámica recurrente. Comparar las respuestas ante una entrada simple.

```

clc ; clear all ; close all; nnet.guis.closeAllViews()
% Vector de entrada
p1 = { 0 0 2 2 2 2 2 -2 -2 -2 -2 0 0 0 0 0 0 0 } ;
%% Construimos la red estática
net1 = linearlayer;
net1.inputs{1}.size = 1;
net1.layers{1}.dimensions = 1;
net1.biasConnect = 0;
net1.IW{1,1} = 1 ;
view(net1) ; a1 = net1(p1);
%% Construimos red dinámica
% Utilizamos red dinámica, pero sin conexiones de retroalimentación (red no recurrente)
net1 = linearlayer([0 1]);
net1.inputs{1}.size = 1;
net1.layers{1}.dimensions = 1;
net1.biasConnect = 0;
net1.IW{1,1} = [0.5 0.5];
view(net1) ; a2 = net1(p1);
%% Construimos red dinámica recurrente
% Utilizamos comando narxnet

```

```

net1 = narxnet(0,1,[],'closed');
net1.inputs{1}.size = 1;
net1.layers{1}.dimensions = 1;
net1.biasConnect = 0;
net1.LW{1} = 0.5;
net1.IW{1} = 0.5 ;
view(net1)
a3 = net1(p1);
% Graficamos entradas y salidas
subplot(411);stem(cell2mat(p1),'linewidth',2, 'color','red');
axis tight; title('Entrada')
subplot(412);stem(cell2mat(a1), 'linewidth',2)
axis tight; title('Salida red estática')
subplot(413); stem(cell2mat(a2), 'linewidth',2)
axis tight; title('Salida red dinámica no recurrente')
subplot(414); stem(cell2mat(a3), 'linewidth',2)
axis tight; title('Salida red dinámica recurrente')

```

En la Fig. 85 se muestra la topología de las 3 redes neuronales. En la Fig. 86 se muestra la entrada de las redes y las respuestas. Se obtienen las siguientes conclusiones:

- Analizando la respuesta de la red estática se observa que la salida solo depende del tiempo presente.
- La red dinámica no recurrente tiene memoria. Se observa que la salida es más larga que la entrada, pero solo una cantidad finita igual a 1 que es la cantidad de retardos. Tiene largo finito debido a que no tiene realimentación de la salida. La respuesta depende de la entrada actual y del historial de entradas
- Se observa que las redes dinámicas recurrentes tienen una respuesta más prolongada que las redes dinámicas de retroalimentación. Para redes lineales, las redes dinámicas de alimentación anticipada se denominan respuesta finita al impulso (FIR), porque la respuesta a una entrada de impulso se convertirá en cero luego de una cantidad de tiempo acotada o finita. Las redes dinámicas recurrentes lineales se denominan respuesta infinita al impulso (IIR), porque la respuesta a un impulso puede decaer a cero (para una red estable), pero nunca llegará a ser exactamente igual a cero.

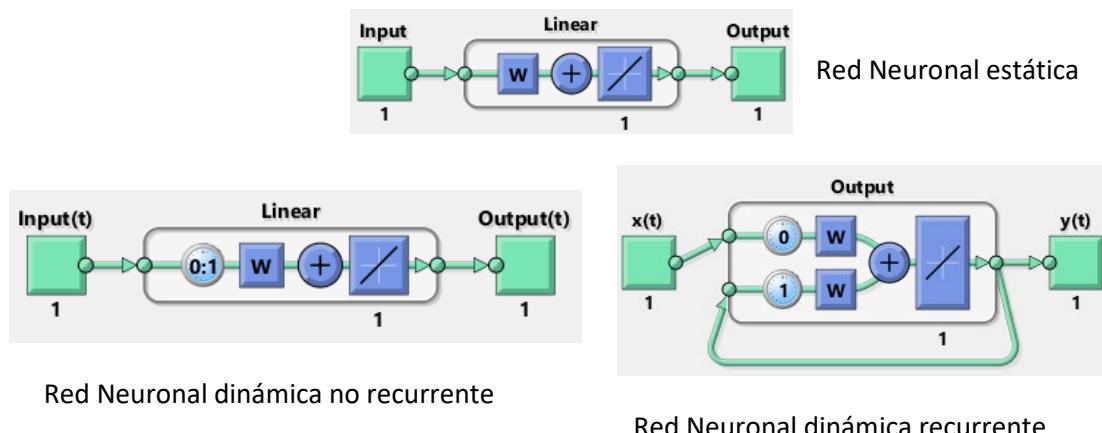


Fig. 85 – Topología de red neuronal estática, dinámica no recurrente y dinámica recurrente

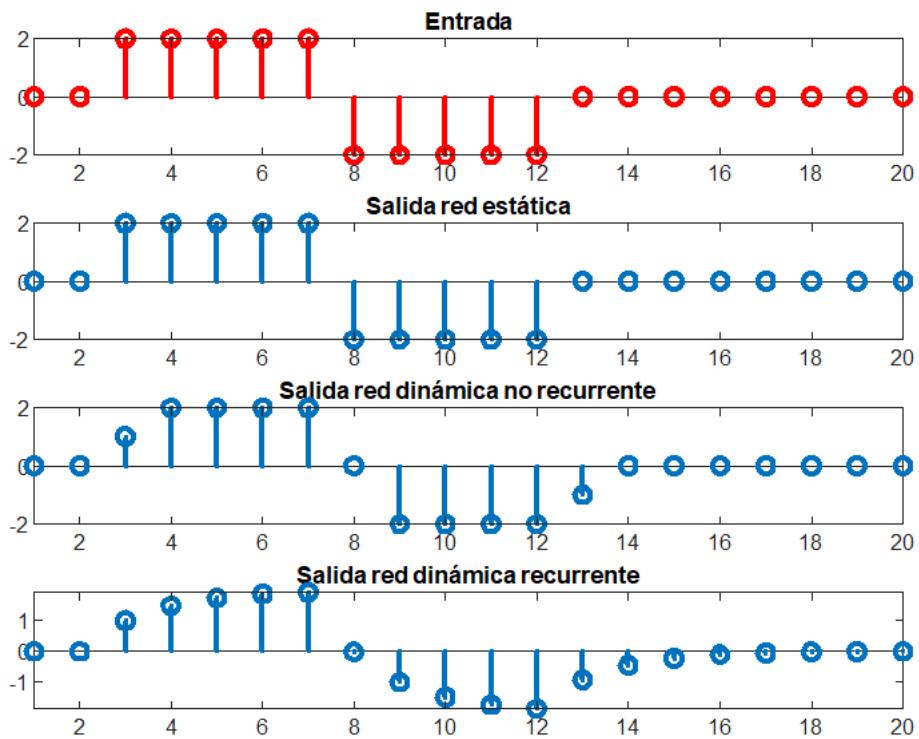
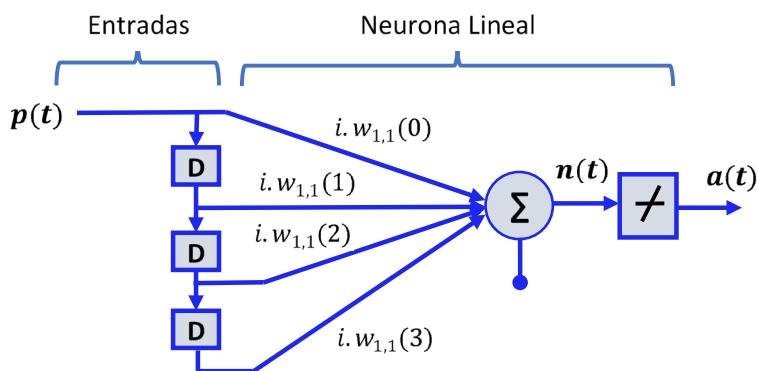


Fig. 86 – Comparación de respuestas para red neuronal estática, dinámica no recurrente y dinámica recurrente

Ejercicio 6.2

Red Neuronal dinámica Adaline

En la Fig. 87 se muestra un ejemplo de red dinámica feedforward Adaline que actúa como filtro. La red tiene una línea de retardos en la entrada, con bloques de retardo identificados con la letra D (en inglés delay).



$$a(t) = i.w_{1,1}(0).p(t) + i.w_{1,1}(1).p(t-1) + i.w_{1,1}(2).p(t-2) + i.w_{1,1}(3).p(t-3)$$

Fig. 87 – Ejemplo de red dinámica feedforward

Para demostrar el funcionamiento de esta red, aplicaremos una onda cuadrada como entrada y estableceremos todos los valores de peso iguales a 1/4:

$$i.w_{1,1}(0) = \frac{1}{4} ; i.w_{1,1}(1) = \frac{1}{4} ; i.w_{1,1}(2) = \frac{1}{4} ; i.w_{1,1}(3) = \frac{1}{4} \quad (6.12)$$

La salida de la red neuronal resulta:

$$a(t) = n(t) = \sum_{d=0}^3 IW(d).p(t-d) \quad (6.13)$$

$$a(t) = n_1(t) = i.w_{1,1}(0).p(t) + i.w_{1,1}(1).p(t-1) + i.w_{1,1}(2).p(t-2) + i.w_{1,1}(3).p(t-3) \quad (6.14)$$

Mediante Matlab® se puede simular la salida correspondiente a una entrada $p(t)$ pulso periódica con el siguiente código:

```
x= [ones(1,8), zeros(1,8), ones(1,8), zeros(1,8)];
n= 0:length(x)-1 ;
b= [1/4 1/4 1/4 1/4] ; y= filter(b,1,x) ;
stem(n,x,'.', 'filled', 'markersize',7); hold on
stem(n,y,'.', 'filled', 'markersize',7); grid on
legend('Entrada','Salida'); ylim([-0.5 1.5])
```

Se muestra la salida en la Fig. 88. Para esta red dinámica, la respuesta en cualquier momento depende de los cuatro valores de entrada anteriores. Si la entrada es constante, la salida se volverá constante después de cuatro pasos de tiempo. Este tipo de red lineal se denomina filtro de respuesta de impulso finito (FIR).

Esta red dinámica tiene memoria. Su respuesta para un momento dado dependerá de la entrada presente y del historial de las entradas. Si la red no tiene ninguna conexión de retroalimentación, entonces solo una cantidad finita de historial afectará la respuesta.

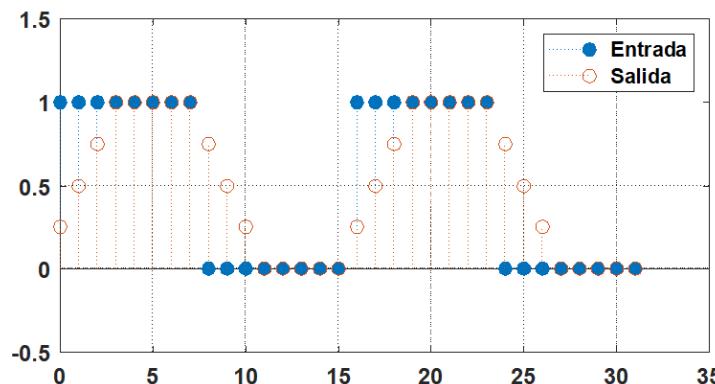


Fig. 88 – Respuesta del ejemplo de red dinámica feedforward tipo FIR

En el siguiente ejemplo, consideramos una red que tiene memoria infinita. Es otra red dinámica lineal simple, pero con una conexión recurrente, como se muestra en la Fig. 89. Se trata de una red dinámica recurrente, la ecuación de funcionamiento de la red resulta:

$$a^1(t) = n^1(t) = IW^{1,1}(0) \cdot p^1(t) + LW^{1,1}(1) \cdot a^1(t-1) \quad (6.15)$$

$$a^1(t) = n^1(t) = iw_{1,1} \cdot p(t) + lw_{1,1}(1) \cdot a(t-1) \quad (6.16)$$

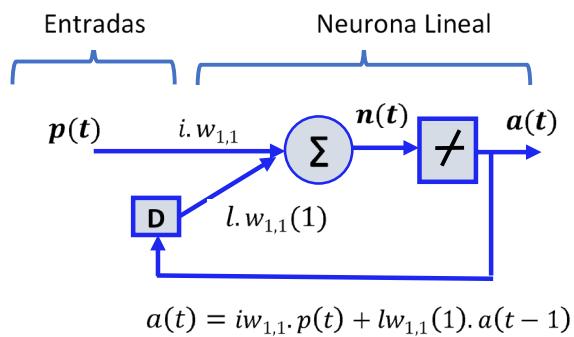


Fig. 89 – Ejemplo de Red dinámica recurrente

Considerando una entrada $p(t)$ pulso periódica, $iw_{1,1} = 1/2$ y $lw_{1,1}(1) = 1/2$ se puede simular la salida de la red. La respuesta se muestra en la Fig. 90. La red responde exponencialmente a un cambio en la secuencia de entrada. A diferencia de la red de filtros FIR, la respuesta de la red en un momento dado es una función del historial infinito de entradas a la red.

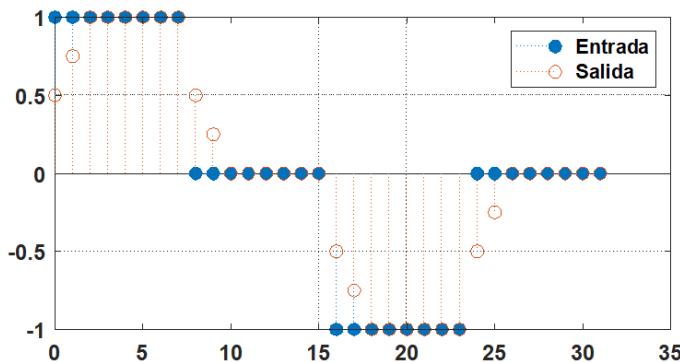


Fig. 90 – Respuesta del ejemplo de red dinámica tipo IIR

Ejercicio 6.3**Red Neuronal no lineal autoregresiva (NAR)**

Ejemplo de una red neuronal no lineal autoregresiva (NAR) utilizada para predecir serie temporal.

La predicción de una secuencia temporal se conoce como predicción de varios pasos. Las redes de circuito cerrado se utilizan para realizar predicciones de varios pasos. Cuando se desconecta la retroalimentación externa, las redes siguen prediciendo mediante un circuito cerrado con retroalimentación interna. En redes tipo NAR, la predicción de los valores futuros de una serie de tiempo se predice solamente a partir de los datos pasados de esa serie.

Se pide analizar el siguiente ejemplo de red NAR en Matlab.

```

%% Ejemplo de red NAR para predecir datos nuevos
% Utilizamos conjunto de datos de Matlab y construimos la red
T1 = simplenar_dataset;
net1 = narnet(1:2, 12);
% Acomodamos los datos con preparets
% Xshift: Entradas desplazadas, Xini: Estados iniciales de retardo de entrada
% Ai: Estados de retardo de la capa inicial, Ts: Objetivos desplazados
[Xshift,Xini,Ai,Ts] = preparets(net1,[],[],T1);
% Entrenamos la red NAR y mostramos su topología
net1 = train( net1,Xshift,Ts,Xini,Ai );
view(net1)
% Calculamos la performance de la red
[Y1,Xf,Af] = net1(Xshift,Xini,Ai);
perf1 = perform(net1,Ts,Y1)
% Primero simulamos la red mediante un lazo cerrado, para luego predecir la salida de
% los siguientes 20 pasos temporales. En esta red solo conecta una entrada con la salida
[netc1,Xini_close,Aic] = closeloop(net1,Xf,Af);
view(netc1)
% Simulamos la red para 20 pasos de tiempo. La entrada es un vector de largo 20
% La red neuronal necesita las condiciones iniciales |Xini_close| y |Aic|.
y2 = netc1(cell(0,20),Xini_close, Aic)
plot(cell2mat(T1), 'linewidth',3); hold on
y = cell2mat(y2) ; n= (length(T1)-length(y))+1 :length(T1) ;
plot(n,y, 'linewidth',3); grid on ;
legend( 'Datos reales ', 'Datos predecidos ' )

```

En la Fig. 91 se muestran los resultados obtenidos con el programa.

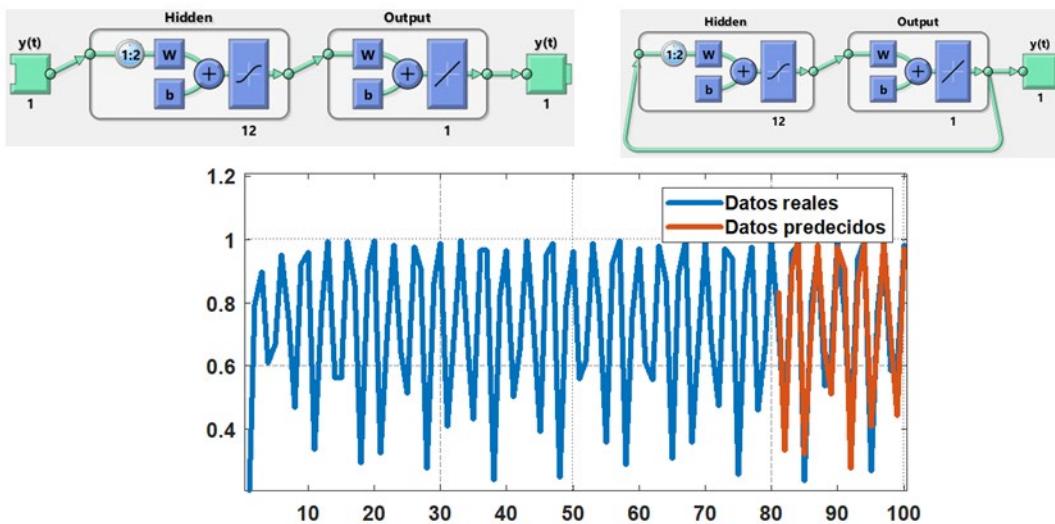


Fig. 91 –Red Neuronal dinámica NAR utilizada para predecir serie temporal. Arriba Izquierda) red con lazo abierto. Arriba derecha) misma red, pero con lazo cerrado. Abajo) Serie de tiempo original y datos predecidos

Ejercicio 6.4

Red Neuronal con retardo temporal (Time Delay)

Ejemplo de red neuronal con retardo temporal (Time Delay) para predecir serie temporal.

Las redes de retardo de tiempo son similares a las redes feedforward, excepto que la entrada tiene una línea de retardo de asociada. Este agregado permite que la red tenga una respuesta dinámica finita a los datos de entrada de series de tiempo. Esta red también es similar a la red neuronal de retardo distribuido (función distdelaynet de Matlab), que tiene retrasos en distintas capas además de la entrada.

```
%% Time Delay Network
% Importamos datos y particionamos
[X1,T1] = simpleseries_dataset;
Xnuevo = X1(81:100);
X1 = X1(1:80); T1 = T1(1:80);
% Entrenamos red Time Delay con 80 observaciones
net1 = timedelaynet(1:2,10);
[ Xshift, Xi, Ai, Ts ] = preparets( net1,X1,T1);
net1 = train(net1,Xshift,Ts,Xi,Ai);
view(net1)
% Calculamos performance
[Y,Xf,Af] = net1(Xshift,Xi,Ai);
perf1 = perform(net1,Ts,Y);
% Arrancamos predicción para 20 pasos temporales con lazo cerrado
[ netc, Xic, Aic ] = closeloop( net1,Xf,Af );
view(netc)
y2 = netc(Xnuevo,Xic,Aic);
```

Ejercicio 6.5**Red Neuronal dinámica autoregresiva con Matlab para serie temporal**

Redes neuronales dinámicas para problemas de serie de tiempo no lineales.

La herramienta ntstool de Matlab permite resolver tres tipos de problemas de series de tiempo no lineales que se muestran en el panel derecho. Para este ejemplo, elegir red neuronal NARX.

- a) Abrir la herramienta de Redes neuronales para series temporales (ntstool). Elegimos la primera opción NARX (autorregresivo)

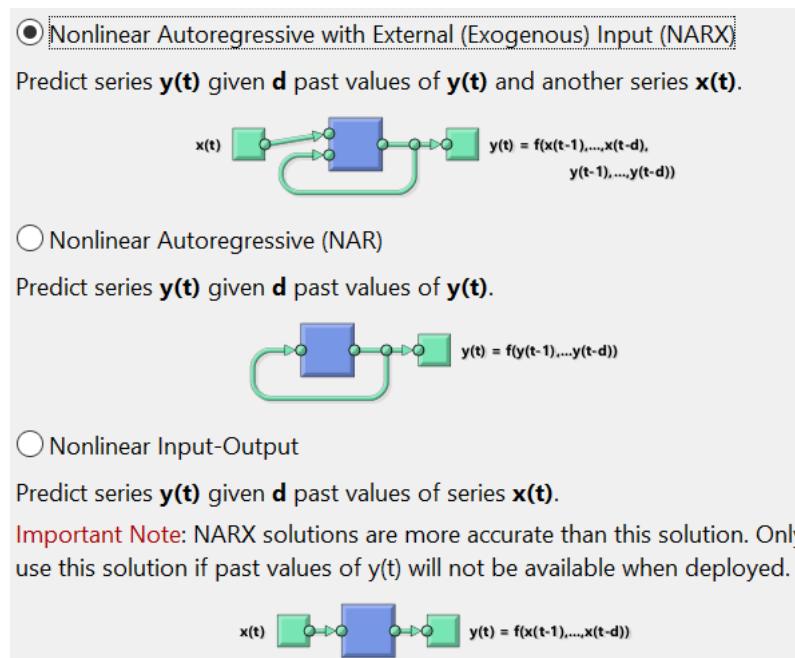


Fig. 92 – Herramienta ntstool de Matlab para redes neuronales con series temporales

Al cargar datos, seleccionar “simplenarx_dataset”. Se pueden cargar datos de ejemplo de Matlab® o cargar datos propios. Elegir 10 neuronas en la capa oculta y 2 retardos. Pulsar el comando “entrenar”. Luego presionar los comandos correspondientes para graficar los resultados.

En la pantalla final generar el código de programa (script), usar el comando “Advanced Script”.

- b) Analizar el funcionamiento del programa generado en el punto anterior.

A continuación, se muestra el programa modificado para una mejor comprensión, a su vez se modifican las primeras líneas para importar los datos correspondientes.

En la Fig. 93 se muestra la topología de las redes utilizadas, y en la Fig. 94 se muestran algunos resultados.

```
% Auto regresión con redes neuronales dinámicas NARX
clc; close all; clear all; nnet.guis.closeAllViews() ;
% Importamos los datos; x entradas, t salidas deseadas
[ X1, T1 ] = simplenarx_dataset;
% Seleccionamos algoritmo de entrenamiento, para ayudas ver help nntrain
% 'trainlm', 'trainbr' o 'trainscg'
trainFcn1 = 'trainlm'; % Algoritmo Levenberg-Marquardt backpropagation.
% Construimos una red No lineal Autoregresiva con entrada externa
feedbackDelays = 1:2 ;
inputDelays = 1:2 ;
neuronas_capa_oculta = 10;
net1 = narxnet(inputDelays,feedbackDelays,neuronas_capa_oculta,'open',trainFcn1);
% Seleccionamos funciones de pre y post procesamiento para las entradas y salidas
% Para mostrar distintas funciones tipear: help nnprocess
% La configuración de la entrada realimentada automáticamente se aplica a la salida realimentada
% Se pueden modificar los parámetros de entrada y salida: net1.inputs{i}.processParam
net1.inputs{1}.processFcns = { 'removeconstantrows', 'mapminmax' } ; %
net1.inputs{2}.processFcns = { 'removeconstantrows', 'mapminmax' } ; %
% Preparamos los datos para entrenamiento y simulación
% La función PREPARETS, prepara los datos de una serie temporal para una red neuronal particular.
% El uso de PREPARETS permite mantener los datos de la serie temporal original sin cambios,
% mientras que los personaliza fácilmente para redes con diferentes números de retrasos, con
% modos de retroalimentación de bucle (lazo) abierto o de lazo cerrado. Ver help prepares.
[ x,xi,ai,t ] = prepares( net1,X1,[],T1 ) ;
% Elegimos porcentaje datos de entrenamiento (70%), validación (15%) y testeo (15%)
% Para ayudas tipear en consola de Matlab: help nndivision
net1.divideFcn = 'dividerand'; % Divide datos en forma aleatoria
net1.divideMode = 'time'; % Divide cada muestra
net1.divideParam.trainRatio = 70/100 ;
net1.divideParam.valRatio = 15/100 ;
net1.divideParam.testRatio = 15/100 ;
% Elegimos función de Performance
% Para ayudas tipear: help nnperformance
net1.performFcn = 'mse'; % error cuadrático medio
% Elegimos las funciones a graficar
% Para ayudas tipear: help nnplot
net1.plotFcns = { 'plotperform', 'plottrainstate', 'plotinerrcorr', 'ploterrhist', ...
    'plotregression', 'plotresponse', 'ploterrcorr' } ;
% Entrenamos la red
[ net1, tr ] = train(net1,x,t,xi,ai);
% Testeamos la red
y = net1(x,xi,ai);
e1 = gsubtract(t,y);
performance1 = perform(net1,t,y)
% Recalculamos Performance de Entrenamiento, Validación y testeo
trainTargets1 = gmultiply(t,tr.trainMask);
valTargets1 = gmultiply(t,tr.valMask);
testTargets1 = gmultiply(t,tr.testMask);
trainPerformance1 = perform(net1,trainTargets1,y)
valPerformance1 = perform(net1,valTargets1,y)
```

```

testPerformance1 = perform(net1,testTargets1,y)
% Mostramos la red
view(net1)
% Graficamos, agregar comentarios si no quiere mostrar todos los resultados
figure, plotperform(tr) ; figure, plottrainstate(tr); figure, ploterrhist(e1)
figure, plotregression(t,y); figure, plotresponse(t,y); figure, ploterrcorr(e1)
figure, plotinerrcorr(x,e1)
% Red de circuito cerrado
% Utilizamos esta red para realizar predicciones de varios pasos. La función CLOSELOOP reemplaza
% la entrada de retroalimentación con una conexión directa desde la capa de salida.
netc1 = closeloop(net1);
netc1.name = [net1.name ' - Lazo cerrado'];
view(netc1)
[ xc, xic, aic, tc ] = preparets( netc1,X1,[],T1);
yc = netc1(xc,xic,aic);
closedLoopPerformance = perform(net1,tc,yc)
% Predicción multipaso
% A veces es útil simular una red en forma de bucle abierto mientras haya datos de salida
% conocidos, y luego cambiar a la forma de bucle cerrado para realizar predicciones de varios pasos
% mientras se proporciona solo la entrada externa. Acá, todos los pasos de tiempo de la serie de
% entrada y la serie de destino, excepto 5, se utilizan para simular la red en forma de bucle abierto,
% aprovechando la mayor precisión que produce la serie de destino:
numTimesteps = size( x, 2 );
knownOutputTimesteps = 1 : (numTimesteps-5) ;
predictOutputTimesteps = ( numTimesteps - 4 ):numTimesteps;
X2 = X1(:,knownOutputTimesteps);
T2 = T1(:,knownOutputTimesteps);
[ x1, xio, aio ] = preparets( net1,X2,[],T2);
[y1,xfo,afo] = net1(x1,xio,aio);
% A continuación, la red y sus estados finales se convertirán a la forma de circuito cerrado para
% hacer cinco predicciones con solo las cinco entradas proporcionadas.
xx2 = X1(1,predictOutputTimesteps);
[ netc1, xic, aic ] = closeloop( net1,xfo,afo ) ;
[y2,xfc,afc] = netc1(xx2,xic,aic);
multiStepPerformance1 = perform(net1,T1(1,predictOutputTimesteps),y2 ) ;
% Se pueden realizar predicciones alternativas para diferentes valores de xx2, o se pueden realizar
% predicciones adicionales continuando la simulación con entradas externas adicionales y los
% últimos estados de ciclo cerrado xfc y afc.
% Red de predicción progresiva
% Para algunas aplicaciones, es útil obtener la predicción con anticipación. La red original devuelve
% y (t + 1) predicho al mismo tiempo que se le da y (t + 1). Para algunas aplicaciones, como la toma
% de decisiones, sería útil haber predicho y (t + 1) una vez que y (t) esté disponible, pero antes de
% que ocurra la y (t + 1) real. Se puede hacer que la red devuelva su salida un paso de tiempo antes
% eliminando un retraso de modo que su retraso de tap mínimo sea ahora 0 en lugar de 1.
% La nueva red devuelve las mismas salidas que la red original, pero las salidas se desplazan a la
% izquierda un paso de tiempo.
nets = removedelay(net1);
nets.name = [net1.name ' - Predict One Step Ahead'];
view(nets)
[xs,xis,ais,ts] = preparets(nets,X1,[],T1);

```

```

ys = nets(xs,xis,ais);
stepAheadPerformance = perform(nets,ts,ys)
% Desarrollo, generamos función de red neuronal
genFunction(net1,'NeuralNetworkFunction_1');
y = NeuralNetworkFunction_1(x,xi,ai);
% Generamos código de función de red para entradas matriciales (no acepta array de celdas) %
genFunction( net1, 'NeuralNetworkFunction_2','MatrixOnly','yes');
x1 = cell2mat(x(1,:)); xx2 = cell2mat(x(2,:));
xi1 = cell2mat(xi(1,:)); xi2 = cell2mat(xi(2,:));
y = NeuralNetworkFunction_2(x1,xx2,xi1,xi2);
% Generamos diagrama Simulink
% gensim(net1);

```

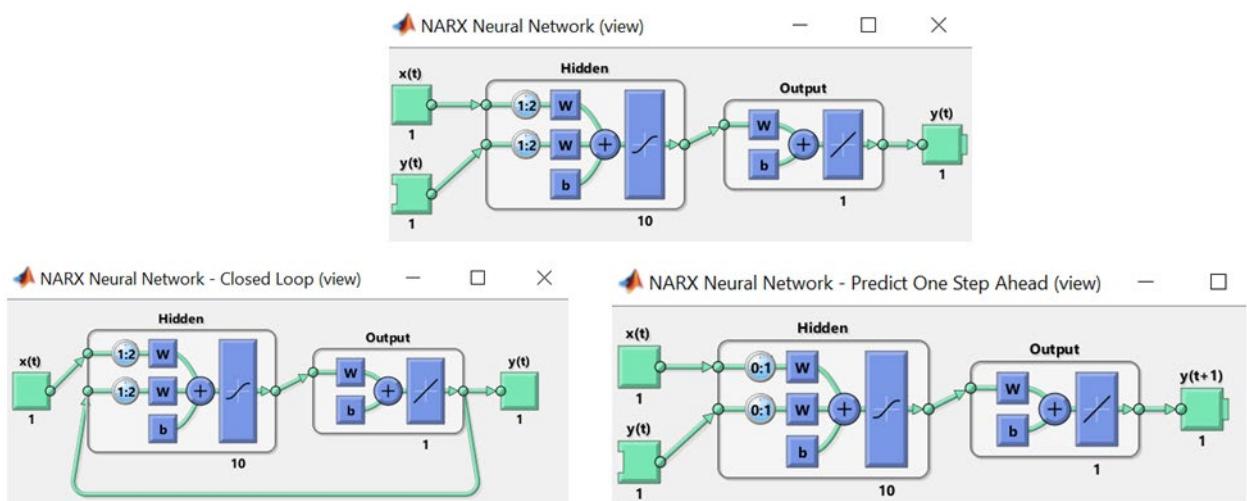


Fig. 93 – Conexiones y topología de redes neuronales dinámicas para el Análisis de las series temporales

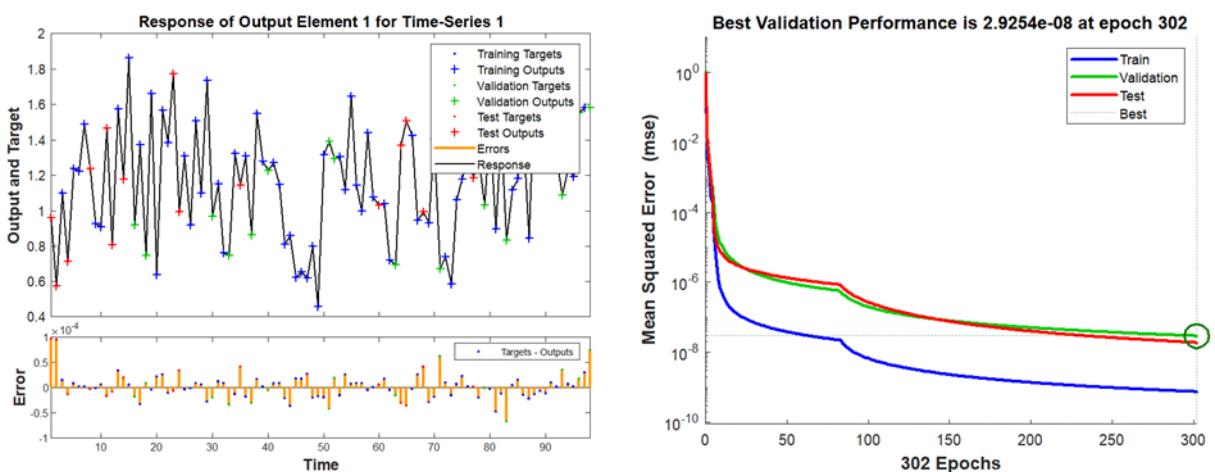


Fig. 94 – Resultados del Análisis de series temporales con redes neuronales dinámicas

Ejercicio 6.6**Ejemplo de Red Neuronal Recurrente (RNN)**

Las RNN o redes neuronales recurrentes son parecidas a las redes con alimentación hacia adelante (conocidas como feedforward), excepto que al menos una capa tiene una conexión de realimentación recurrente con un retraso asociado. Con esta conexión la red tiene una respuesta dinámica infinita a los datos de entrada de series de tiempo. Estas redes son semejantes a las redes neuronales de retardo de tiempo y a las redes con retardo distribuido. En la Fig. 95 se muestra un ejemplo de RNN con la topología de red utilizada.

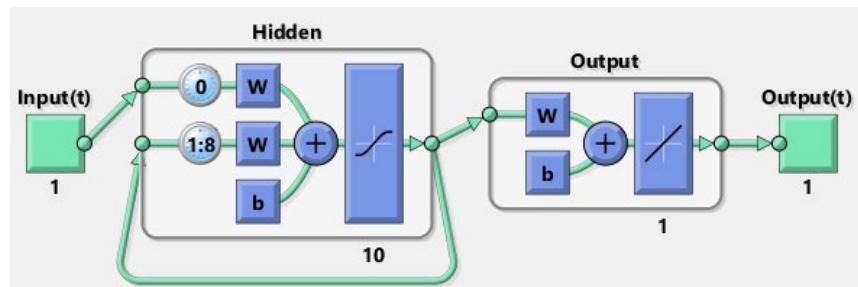


Fig. 95 – Topología de red neuronal recurrente RNN con 8 retardos y 10 neuronas ocultas

A continuación, se muestra un código de ejemplo para de ajuste datos de una serie temporal con redes neuronales recurrentes (RNN) en el entorno Matlab®. Se pide verificar el funcionamiento del programa y obtener conclusiones.

```
clear all; close all; clc; nnet.guis.closeAllViews();
%% Armamos el conjunto de datos
% También se pueden importar los datos
% load mis_datos2 ; % siendo x: entradas, t: salidas deseadas
x=0; rng(1)
for i=1:120
    dx = abs(0.01 *randn());
    x=[x x(end)+ dx];
end
% t son las salidas deseadas, suma de señales
t= cos(4*x) + sin(20*x) +1;
t= t +0.21 *randn(1,length(t)) ; % Agregamos ruido
figure; plot(x,t, 'linewidth',2); axis tight; grid on
X1 =num2cell(x) ; T1=num2cell(t) ;
%% Construimos RNN con 8 retardos y con 10 neuronas en la capa oculta.
net1 = layrecnet(1:8, 10) ;
% función prepares, acomoda las series temporales de entrada y destino
% para simular o entrenar la red
net1.trainParam.epochs =600 ;
[Xshift,Xi,Ai,Ts] = prepares(net1, X1, T1); % acomodamos los datos
[net1, tr1] = train(net1, Xshift, Ts, Xi, Ai) ; % Función train entrena la red
view(net1) %% Mostramos la red
Y = net1(Xshift, Xi, Ai) ;
```

```

perf1 = perform(net1, Y, Ts)
%% Graficamos los resultados
figure; plotperform(tr1);
% net1.plotFcns={'plotperform','plottrainstate', 'plotinerrcorr', 'ploterrhist', ...
%   'plotregression', 'plotresponse', 'ploterrcorr'};
% Graficamos, agregar comentarios si no se quiere mostrar todos los resultados
figure, plottrainstate(tr1);
t1 = t(9:end) ;
figure, plotregression(t1,Y);
figure, plotresponse(Ts,Y);
E = gsubtract(Ts,Y);
figure; ploterrhist(E)
figure ; ploterrcorr(E) ; %figure; plotinerrcorr(Xshift,E)

```

En la Fig. 96 se muestran los resultados del programa. Se pueden obtener los gráficos a través de la herramienta nntraintool de Matlab® o con código de programa. Para los datos de entrenamiento obtenemos un alto valor del coeficiente de correlación, $R = 1$

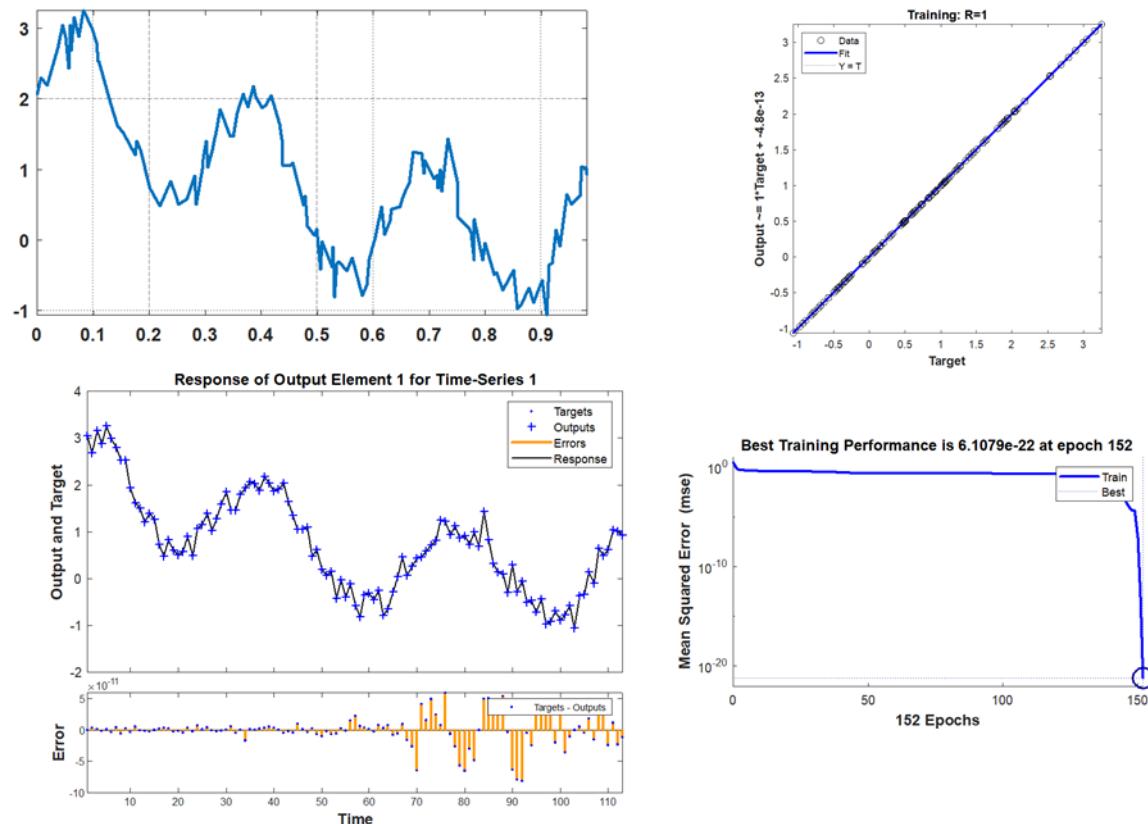


Fig. 96 – Resultados del Ajuste de datos con redes neuronales recurrentes

Ejercicio 6.7**Predicción de señales temporales mediante una RNN tipo LSTM**

En este ejemplo mediante una red neuronal se predicen los valores de una serie temporal. Se utiliza una RNN del tipo LSTM con un largo de secuencia de 4 valores, con 1 entrada, 1 capa oculta de tamaño 2 y una capa lineal. Se implementa con la siguiente función:

```
LSTM( (lstm): LSTM(1, 2, batch_first=True)
      (fc): Linear(in_features=2, out_features=1, bias=True) )
```

En la Fig. 97 se muestran los resultados, la RNN se entrena solo con la primera mitad de los datos desde 0 hasta 122 (indicado con línea roja). A continuación, se muestra el código completo utilizado.

```
import numpy as np ## numpy
import matplotlib.pyplot as plt ## pyplot
import torch ##
import torch.nn as nn ##
from torch.autograd import Variable
from sklearn.preprocessing import MinMaxScaler
## Generamos Datos de entrenamiento y graficamos
tt = np.arange(0, 250, 1)
dd = np.sin(0.2*tt) + 1.5* np.sin(0.05*tt)+ np.random.normal(scale=0.15, size=len(tt))
set_entren = dd.reshape(-1, 1)
plt.figure(figsize=(8, 4))
plt.plot(set_entren, label = 'Datos') ; plt.show()
## Escalado de datos
escalado1 = MinMaxScaler()
datos_entren = escalado1.fit_transform(set_entren)
## Generamos ventanas con datos desplazados
def genero_ventanas_desplazadas(data, largo_seq):
    xx = [] ; yy = []
    for i in range( len( data ) - largo_seq-1):
        _xx = data[i:(i+largo_seq)]
        _yy = data[i+largo_seq]
        xx.append(_xx)
        yy.append(_yy)
    return np.array(xx),np.array(yy)
largo_seq = 4
x, y = genero_ventanas_desplazadas(datos_entren, largo_seq)
## Separamos en entrenamiento y test
largo_entren = int(len(y) * 0.5)
largo_test = len(y) - largo_entren
datosX = Variable(torch.Tensor(np.array(x)))
datosY = Variable(torch.Tensor(np.array(y)))
testX = Variable(torch.Tensor(np.array(x[largo_entren:len(x)])))
testY = Variable(torch.Tensor(np.array(y[largo_entren:len(y)])))
entrenX = Variable(torch.Tensor(np.array(x[0:largo_entren])))
```

```

entrenY = Variable(torch.Tensor(np.array(y[0:largo_entren])))
## Generamos Modelo LSTM
class LSTM(nn.Module): ## clase
    def __init__(self, cant_clases, largo_n_entrada, largo_n_ocultas, cant_capas):
        super(LSTM, self).__init__()
        self.num_classes = cant_clases
        self.num_layers = cant_capas
        self.input_size = largo_n_entrada
        self.hidden_size = largo_n_ocultas
        self.seq_length = largo_seq
        self.lstm = nn.LSTM(input_size=largo_n_entrada, hidden_size=largo_n_ocultas,
                            batch_first=True, num_layers=cant_capas)
        self.fc = nn.Linear(largo_n_ocultas, cant_clases)
    def forward(self, x1):
        h_0 = Variable(torch.zeros(self.num_layers, x1.size(0), self.hidden_size) )
        c_0 = Variable(torch.zeros(self.num_layers, x1.size(0), self.hidden_size) )
        # Propagamos entradas a través de la red LSTM
        ula, (h_out1, _) = self.lstm(x1, (h_0, c_0))
        h_out1 = h_out1.view(-1, self.hidden_size)
        out1 = self.fc(h_out1)
        return out1
##### Entrenamos el modelo
# Parámetros
largo_n_entrada = 1 ; largo_n_ocultas = 2
taza_aprendizaje = 0.01 ; cant_iteraciones = 2500
cant_capas = 1 ; cant_clases = 1
lstm1 = LSTM(cant_clases, largo_n_entrada, largo_n_ocultas, cant_capas)
print(lstm1)
criterio1 = torch.nn.MSELoss() # error cuadrático medio en la regresión
#optimizador = torch.optim.SGD(lstm1.parameters(), lr=taza_aprendizaje)
optimizador = torch.optim.Adam(lstm1.parameters(), lr=taza_aprendizaje)
# Lazo de entrenamiento
for iteracion in range(cant_iteraciones):
    salidas = lstm1(entrenX)
    optimizador.zero_grad()
    # obtain the loss function
    loss = criterio1(salidas, entrenY)
    loss.backward()
    optimizador.step()
    if iteracion % 100 == 0:
        print("Iteración: %d, loss: %1.6f" % (iteracion, loss.item()))
##### Testeo
lstm1.eval()
entren_predic = lstm1(datosX)
data_predict = entren_predic.data.numpy()
datosY_plot = datosY.data.numpy()
data_predict = escalado1.inverse_transform(data_predict)
datosY_plot = escalado1.inverse_transform(datosY_plot)
##### Graficamos
plt.rcParams.update({'font.size': 14})

```

```

plt.figure(figsize=(8, 4))
plt.axvline(x=largo_entren, c='r', linewidth=2, linestyle='--') # Línea de separación
plt.plot(datosY_plot, label='Salida original')
plt.plot(data_predict, label='Salida predecida') ; plt.legend()
plt.suptitle('Serie Temporal Predecida')
plt.show() ; plt.savefig('Resultado.png')

```

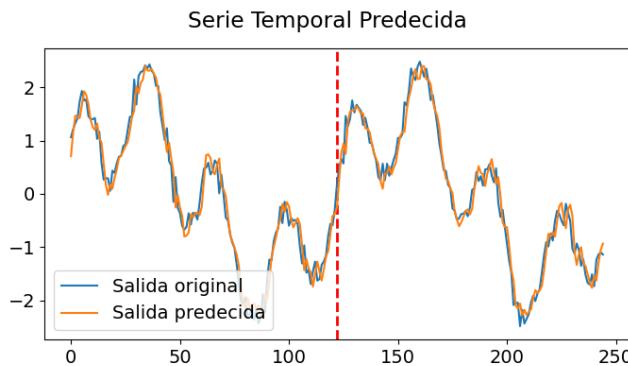


Fig. 97 – Resultados de predicción de serie temporal con RNN tipo LSTM

Ejercicio 6.8

Traductor de inglés a castellano con modelo de transformación secuencial

En este ejemplo se utiliza un modelo secuencial con el módulo Keras de Tensorflow. Este modelo aprende de un diccionario de inglés a castellano con 118964 oraciones cortas de distinto largo. El programa vectoriza texto usando la capa Keras TextVectorization, implementa una capa TransformerEncoder, una capa TransformerDecoder y una capa PositionalEmbedding. El modelo utiliza 19.960.216 parámetros, todos entrenables. Una vez entrenado el algoritmo traduce sin utilizar el diccionario original. A continuación, se muestran solo algunos resultados.

Tom is the happiest person in the world right now. ----> Tom es la persona más feliz del mundo.

I should've told you. ----> debería haber dicho

I know it for sure. ----> lo sé por qué está seguro

I'm sorry about last night. ----> lamento lo anoche



Solicitar código al siguiente email:

Ejercicio 6.9

Generación de texto con redes neuronales recurrentes (RNN) del tipo GRU

En este código se toma un texto, que puede ser una obra literaria u otro texto, y se entrena una RNN del tipo GRU. En base a este entrenamiento el software genera un texto nuevo, pero con las mismas características de escritura que el texto original. El modelo utiliza caracteres, por lo que aprende a crear palabras significativas a partir de personajes y agrega palabras relacionadas entre sí. Repasa miles de palabras y aprende la relación entre diferentes caracteres y cómo se utilizan para crear palabras significativas. Luego replica esto y nos devuelve oraciones con palabras significativas. Resulta interesante generar texto para sonar como la obra de Shakespeare o intentando rimar como el Dr. Seuss.

En la Tabla IX se muestran las capas de la RNN GRU secuencial utilizada.

Tabla IX – RNN GRU utilizada para generar texto

Capa	Tamaño de Salida	Parámetros
embedding (Embedding)	(64, None, 256)	16640
gru (GRU)	(64, None, 1024)	3938304
dense (Dense)	(64, None, 65)	66625
Cantidad total de parámetros: 4.021.569		

Los resultados del algoritmo se deben mejorar con un corrector ortográfico. A continuación, se muestran algunos resultados de generar texto nuevo a partir de la obra de Shakespeare.

QUEEN: So, it's not good at all. I have contemplated your eyes; we will see the opposite. Tell him when the duke was king.

And who should say, I will wish him too?

TRANIO: How is daughter fighting now? Oh, being someone else's. She unlocks a field of reminders. Like you, real boy, buty: if nothing shoots a minute, that layer is limit gold.

ROMEO: Goodbye Sir! you?

CORIOLANUS: You will not do it.

Ver descarga de código al final de este capítulo

Ejercicio 6.10

Clasificación de textos mediante el análisis sentimental con RNN bidireccional LSTM

En este ejemplo se predice el sentimiento de los comentarios de películas mediante clasificación de texto. Se utiliza procesamiento natural del lenguaje (PNL) para predecir el sentimiento de las críticas de películas como positivo o negativo con librerías de Keras en Python.

Se desarrolla y evalúa un modelo de percepción multicapa con red neuronal convolucional unidimensional. Se utiliza un RNN bidireccional LSTM con 3.269.761 parámetros.

En la Fig. 98 se muestra un modelo simplificado con solo 1 capa bidireccional, este modelo se genera mediante el siguiente código:

```
model = tf.keras.Sequential([
    encoder, tf.keras.layers.Embedding(
        input_dim=len(encoder.get_vocabulary()), output_dim=64,
        # Utilizamos enmascaramiento para manejar longitudes de secuencia variable
        mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)])
model.summary()
```

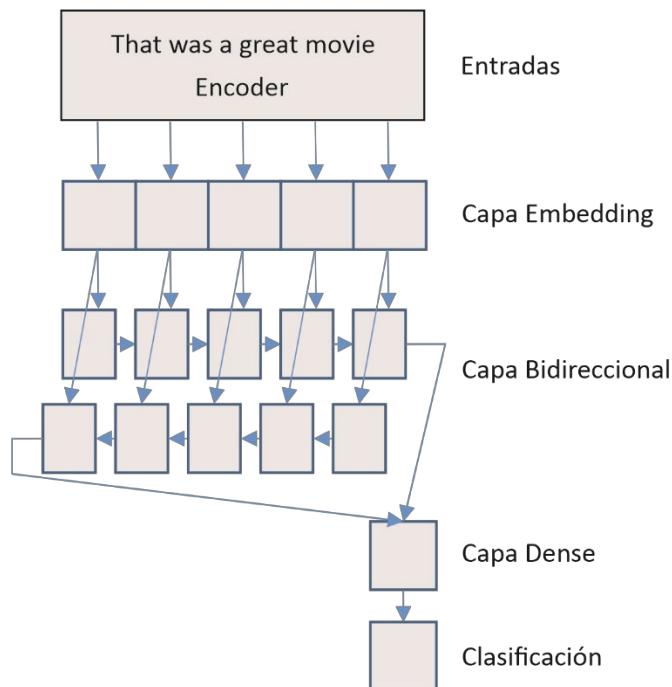


Fig. 98 – Ejemplo de RNN con capa bidireccional

Se utiliza la base de datos IMDB: conjunto de datos de revisión de películas grandes. IMDB contiene 25.000 comentarios clasificados en 2 clases: buenas o malas (comentarios positivos o negativos) para entrenamiento. Para el testeo o prueba también se dispone de la misma cantidad. El problema es determinar si un comentario tiene un sentimiento negativo o positivo. Los datos fueron recopilados y armados por investigadores de Stanford y se utilizan en muchas publicaciones. En la publicación de Maas Andreew (2011) se utilizó una división del 50/50 de los datos para entrenamiento y prueba. Se logró una precisión del 88,89%. Los datos también se utilizaron como base para una competencia de Kaggle a principios de 2015. Se logró una precisión superior al 97%.

Se muestran 5 películas ordenadas por su calificación

- 1 - Sueño de Libertad (1994) - Starring: Frank Darabont (director), Tim Robbins, Morgan Freeman
9.220458798204707
- 2 - El Padrino (1972) - Starring: Francis Ford Coppola (director), Marlon Brando, Al Pacino
9.14725023905667
- 3 - El Padrino 2ª Parte (1974) - Starring: Francis Ford Coppola (director), Al Pacino, Robert De Niro
8.980486406031105
- 4 - Batman - El caballero de la noche (2008) - Starring: Christopher Nolan (director), Christian Bale, Heath Ledger
8.972434875191393
- 5 - 12 hombres en pugna (1957) - Starring: Sidney Lumet (director), Henry Fonda, Lee J. Cobb
8.938290080259588

Programa:

```
import numpy as np ## numpy
from tensorflow.keras import layers # capas ##
from tensorflow import keras # Módulo Keras
from tensorflow.keras.datasets import imdb #Base de datos IMDB
largo_max = 250 # Solo consideramos las primeras 250 palabras de cada película
max_caracteristicas = 24000 # Solo consideramos las primeras 24000 palabras más importantes
#### Cargamos base de datos IMDB establecemos misma longitud
(x_entren, y_entren), (x_val, y_val) = imdb.load_data( num_words=max_caracteristicas )
x_entren = keras.preprocessing.sequence.pad_sequences(x_entren, maxlen=largo_max)
x_val = keras.preprocessing.sequence.pad_sequences(x_val, maxlen=largo_max )#
print(len(x_entren), "Secuencias de entrenamiento")
print(len(x_val), "Secuencias de Validación")
#### Generamos el Modelo
# Entrada para secuencias de enteros de longitud variable
entradas = keras.Input(shape=(None,), dtype="int32")
# Incrustamos cada entero en un vector de largo 128
vect = layers.Embedding(max_caracteristicas, 128)(entradas)
# Agregamos 2 LSTMs bidireccionales y un clasificador Dense
vect = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(vect)
vect = layers.Bidirectional(layers.LSTM(64))(vect)
salidas = layers.Dense(1, activation="sigmoid")(vect)
# Agrupamos capas en un objeto con funciones de entrenamiento e inferencia.
```

```

model = keras.Model(entradas, salidas)
##### Mostramos modelo
model.summary()
##### Entrenamos y evaluamos el modelo
model.compile("adam", "binary_crossentropy", metrics=["accuracy"] )
iterac = 4
model.fit(x_entren, y_entren, batch_size=32, epochs=iterac, validation_data=(x_val,y_val))
### Agregado
##### Evaluamos entrenamiento
## scores = model.evaluate( X_test, y_test, verbose = 0 )
scores1 = model.evaluate(x_entren, y_entren, verbose=0)
print("Precisión del modelo para el conjunto de datos IMDB: {:.3f}%".format(scores1[1]*100))
# Predecimos una muestra
Xnuevo = x_entren[120]
ynuevo = model.predict(Xnuevo)
##### Mostramos algunas palabras de IMDB
(x_entren, y_entren), (x_test, y_test) = imdb.load_data()
# Obtenemos 3 textos en forma aleatoria
for kk in np.random.randint(0, len(x_entren), 3):
    INDICE_INI=3 # offset
    word_to_numberID = imdb.get_word_index()
    word_to_numberID = {ii:(INDICE_INI+jj) for ii,jj in word_to_numberID.items()}
    word_to_numberID["<PAD>"] = 0
    word_to_numberID["<INICIO >"] = 1
    word_to_numberID["<UNK>"] = 2
    word_to_numberID["<UNUSED>"] = 3
    id_to_word = {value:key for key,value in word_to_numberID.items()}
    print(' '.join(id_to_word[id1] for id1 in x_entren[kk]))
    print('Respuesta: ', y_entren[kk])

```

Se muestran solo algunos resultados del programa:

Precisión del modelo para el conjunto de datos IMDB: 98.968%

<INICIO > this is an excellent movie with a stellar cast and some great acting i never tire of watching it i especially love the scene where danny glover's character and kevin kline's character namely ...

Respuesta: 1 (Positivo)

<INICIO > i have recently watched this movie twice and i can't seem to understand why the h ll the makers made this pile of crap i mean yes it gives a great impression of hitler's environment and i mean the way they reproduced austria in the late 1890's wwi and the inter war period what i can't understand is why they pictured ...

Respuesta: 0 (Negativo)

A su vez, mediante la base de datos *reuters* se pueden clasificar los comentarios en 46 tópicos: cocoa, grain, veg-oil, earn, acq, wheat, copper, etc.

Ver descarga del código completo al final de este capítulo

Ejercicio 6.11**Reconocimiento de acciones para clasificación de videos con red neuronal RNN GRU combinada con CNN**

En este ejemplo se utiliza la base de datos UCF101 compuesta de videos cortos. Se entrena una red neuronal para entrenar un clasificador de videos. El conjunto de datos contiene videos separados en distintas categorías según las acciones realizadas, tales como afeitarse, tiro de críquet, tenis de mesa, puñetazos, andar en bicicleta, etc. Los videos consisten en una secuencia ordenada de fotogramas con información espacial y temporal, la red neuronal tiene en cuenta esta información. En la Fig. 99 se muestra un ejemplo de una imagen de un video de muestra. Se utiliza una arquitectura RNN con capas GRU combinada con red neuronal convolucional (CNN), se muestran los detalles de las capas en la Tabla X. Se utiliza el siguiente código para generar el modelo:

```
MAX_SEQ_LENGTH = 25 ; NUM_FEATURES = 2048 # Parámetros
frame_features_input = keras.Input(( MAX_SEQ_LENGTH, NUM_FEATURES )) #
mask_input = keras.Input(( MAX_SEQ_LENGTH,), dtype="bool" ) #
x = keras.layers.GRU( 16, return_sequences= True)(
    frame_features_input, mask= mask_input )
x = keras.layers.GRU( 8 )(x ) # GRU
x = keras.layers.Dropout(0.4)(x) # Dropout
x = keras.layers.Dense(8, activation="relu")(x) # Dense
output = keras.layers.Dense(len(class_vocab), activation="softmax")(x)
rnn_model = keras.Model([frame_features_input, mask_input], output)
```

Tabla X – Detalles de la red RNN GRU combinada con CNN

Capa	Tamaño	Parámetros	Conectada a capa
input_3 (InputLayer)	[(None, 25, 2048)]	0	
input_4 (InputLayer)	[(None, 25)]	0	
gru (GRU)	(None, 25, 16)	99168	input_3[0][0] y input_4[0][0]
gru_1 (GRU)	(None, 8)	624	gru[0][0]
dropout (Dropout)	(None, 8)	0	gru_1[0][0]
dense (Dense)	(None, 8)	72	dropout[0][0]
dense_1 (Dense)	(None, 5)	45	dense[0][0]
Cantidad total de parámetros: 99909			

Dado que un video es una secuencia ordenada de fotogramas, se puede simplemente extraer los fotogramas y ponerlos en un tensor 3D. Pero la cantidad de fotogramas puede diferir de un video a otro, con lo que habría que llenar. Como alternativa, en este ejemplo se guardan fotogramas de video en un intervalo fijo hasta que se alcanza un recuento máximo de fotogramas. En el caso de que el recuento de fotogramas de un video sea menor que el recuento máximo de fotogramas, llenaremos el video con ceros. Este flujo de trabajo es idéntico a los problemas que involucran secuencias de texto. Se sabe que los videos del conjunto de datos UCF101 no contienen variaciones

bruscas en los objetos y acciones en los finales. Debido a esto, se considera solo unos pocos fotogramas para la tarea de aprendizaje. Pero este enfoque puede no generalizarse bien a otros problemas de clasificación de videos. Usaremos el método VideoCapture () de OpenCV para leer fotogramas de videos.

El módulo de Keras de TensorFlow proporciona una serie de modelos de red previamente entrenados de última generación, utilizaremos el modelo InceptionV3 para este propósito. Las etiquetas de los videos son cadenas, por lo que deben convertirse a alguna forma numérica antes de que se envíen al modelo de red neuronal. Aquí usaremos la capa StringLookup para codificar las etiquetas de clase como enteros.

El clasificador nos devuelve las probabilidades de que el video a clasificar represente diferentes acciones. Se muestra solo un ejemplo de clasificación del programa:

Punch: 76.17%

CricketShot: 11.61%

ShavingBeard: 11.07%

PlayingCello: 0.84%

TennisSwing: 0.30%



Fig. 99 – Ejemplo de imagen de video para clasificar acciones



Ver descarga del código al final de este capítulo. Ante consultas escribir al siguiente email:

Ejercicio 6.12

Reconocimiento de acciones para clasificación de videos con red neuronal I3D CNN. Comparación de distintas arquitecturas

En este ejemplo se reconocen acciones en videos mediante el modelo Kinetics 400, este modelo se puede descargar desde la siguiente página: tfhub.dev/deepmind/i3d-kinetics-400/1. El modelo se describe en el artículo Carreira, 2017. Este modelo introdujo una nueva arquitectura para la clasificación de video, *Inflated 3D Convnet (I3D)* con 79 millones de parámetros. Esta arquitectura logró resultados de vanguardia en los conjuntos de datos UCF101 y HMDB51 al ajustar estos modelos. En la publicación se muestra un análisis sobre las arquitecturas actuales en la tarea de clasificación de acciones (Carreira,2017). También se muestra un análisis y comparación con la nueva red ConvNet 3D inflada de dos flujos (I3D) que se basa en la inflación de ConvNet 2D con filtros y núcleos de agrupación de clasificación de imágenes muy profundos. Las ConvNets se expanden a 3D, lo que hace posible aprender extractores de características espacio temporales sin problemas a partir de video, al mismo tiempo que aprovecha los exitosos diseños de arquitectura de ImageNet e incluso sus parámetros. Se demuestra que los modelos I3D mejoran considerablemente la clasificación, alcanzando un 80,9% de precisión en base de datos HMDB-51 y 98,0% en UCF-101. También se demuestra la transferencia de aprendizaje de un conjunto de datos de videos (Kinetics) hacia otro conjunto de datos (UCF-101 / HMDB-51) para una tarea similar, aunque para diferentes acciones (Carreira,2017).

En este ejercicio el modelo original Kinetics 400 se entrenó previamente con el conjunto de datos kinetics-400 que reconoce 400 acciones diferentes. Las etiquetas para estas acciones se pueden encontrar en el archivo de mapa de etiquetas. Con este modelo, en este ejemplo se reconocen actividades en videos de un conjunto de datos UCF101 que contiene 13320 videos separados en 101 categorías. A modo de ejemplo, se muestran los resultados de clasificar correctamente un video.

playing cricket: 97.77%

skateboarding: 0.71%

robot dancing: 0.56%

roller skating: 0.56%

golf putting: 0.13%

Ver descarga del código al final de este capítulo.

En el artículo de Carreira (2017) se comparan distintas arquitecturas de redes para los conjuntos de datos UFC-101, HMDB-51 y Kinetics, se muestran algunos resultados en la Tabla XI.

Tabla XI – Comparación de performance para distintas arquitecturas con los conjuntos de datos UCF-101 y HMDB-51 (Carreira, 2017). Ver detalles en la publicación

Modelo	UCF-101	HMDB-51
Two Stream	88,0	59,4
IDT	86,4	61,7
Dynamic Image Network + IDT	89,1	65,2
TDD + IDT	91,5	65,9
Two Stream Fusion + IDT	93,5	69,2
Temporal Segment Networks	94,2	69,4
ST-ResNet + IDT	94,6	70,3
Deep Networks, Sports 1 M pre-training	65,2	-
C3D one network, Sports 1 M pre-training	82,3	-
C3D ensemble, Sports 1 M pre-training	85,2	-
C3D ensemble + IDT, Sports 1 M pre-	90,1	-
RGB-I3D, Imagenet+Kinetics pre-training	95,6	74,8
Flow-I3D, Imagenet+Kinetics pre-training	96,7	77,1
Two-Stream I3D, Imagenet+Kinetics pre-	98,0	80,7
RGB-I3D, Kinetics pre-training	95,1	74,3
Flow-I3D, Kinetics pre-training	96,5	77,3
Two-Stream I3D, Kinetics pre-training	97,8	80,9

Ejercicio 6.13

Predicción de tramas de video mediante RNN con capas LSTM

En este ejemplo se utiliza la base de datos de videos MNIST. Cada video está compuesto por 20 tramas con 2 dígitos que se desplazan. La arquitectura utilizada en este ejemplo combina el procesamiento de series temporales y la visión por computadora al introducir una celda recurrente convolucional en una capa LSTM. El modelo Convolutional LSTM se utiliza para la predicción del siguiente cuadro, es decir que se predice qué cuadros de video vienen a continuación dada una serie de cuadros pasados.

Tabla XII – Capas del modelo Convolucional tipo LSTM para predecir videos

Capa	Tamaño de Salida	Parámetros
input_1 (InputLayer)	[(None, None, 64, 64, 1)]	0
conv_lst_m2d (ConvLSTM2D)	(None, None, 64, 64, 64)	416256
batch_normalization	(None, None, 64, 64, 64)	256
conv_lst_m2d_1 (ConvLSTM2D)	(None, None, 64, 64, 64)	295168
batch_normalization_1	(None, None, 64, 64, 64)	256
conv_lst_m2d_2 (ConvLSTM2D)	(None, None, 64, 64, 64)	33024
conv3d (Conv3D)	(None, None, 64, 64, 1)	1729
Cantidad total de parámetros:	74689	

En la Fig. 100 se muestra un ejemplo con las 20 tramas de un video. Una vez entrenado el modelo, se toman las primeras 10 tramas de un video y se predicen las próximas 10 tramas, ver Fig. 101. Con estas imágenes se arma un video con las tramas originales y las tramas prediccidas.

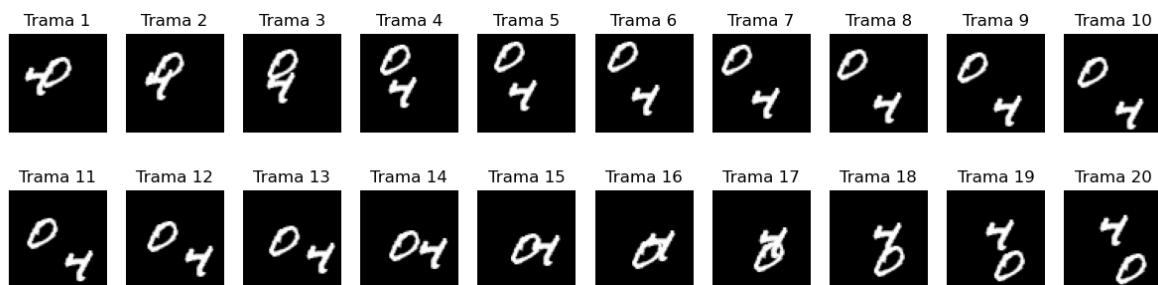


Fig. 100 – Ejemplo de video original separado en 20 tramas



Fig. 101 – Resultados de la predicción de tramas de video, las primeras 10 tramas corresponden a las tramas originales y las últimas 10 corresponden a las tramas predichas

Ver descarga del código al final de este capítulo

Ejercicio 6.14

Clasificación de vocales con redes LSTM mediante herramienta deepNetworkDesigner de Matlab.

En este ejemplo se utiliza el conjunto de datos “Japanese Vowels”, este conjunto contiene voces masculinas que pronuncian 2 vocales japonesas / ae / sucesivamente. Los predictores contienen secuencias de longitud variable. Las salidas son vectores categóricos con etiquetas 1,2, ..., 9.

Se muestran los pasos a seguir para diseñar la red y luego el código asociado.

Escribir en líneas de comandos:

```
>> deepNetworkDesigner
```

Ir a “Sequences Networks”, abrir red pre armada “Sequence-to-Label”, ver Fig. 102.

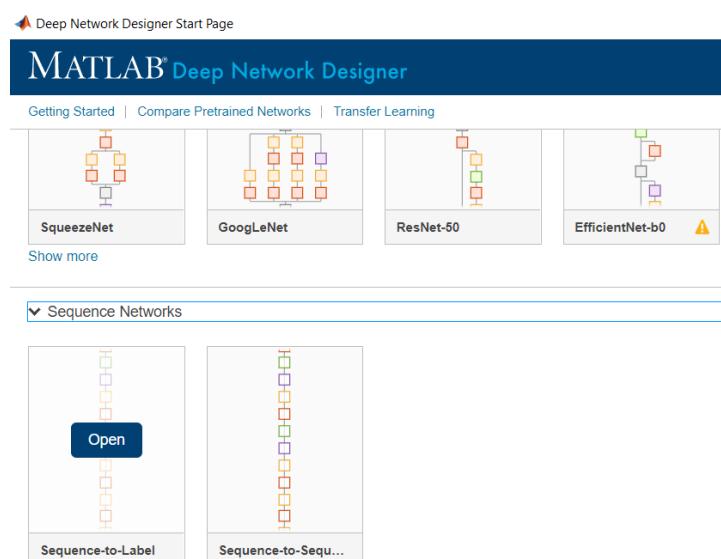


Fig. 102 – Detalles para abrir red pre armada LSTM

Una vez abierta la red, seleccionar “sequenceInputLayer” y verificar que “InputSize” se encuentre configurado en 12 para que coincida la dimensión.

Luego, seleccionar “lstmLayer” e ingresar 110 en “NumHiddenUnits”, ver Fig. 103.

Seleccionar “fullyConnectedLayer” y verificar que “OutputSize” se encuentre configurado en 9 (número de clases).

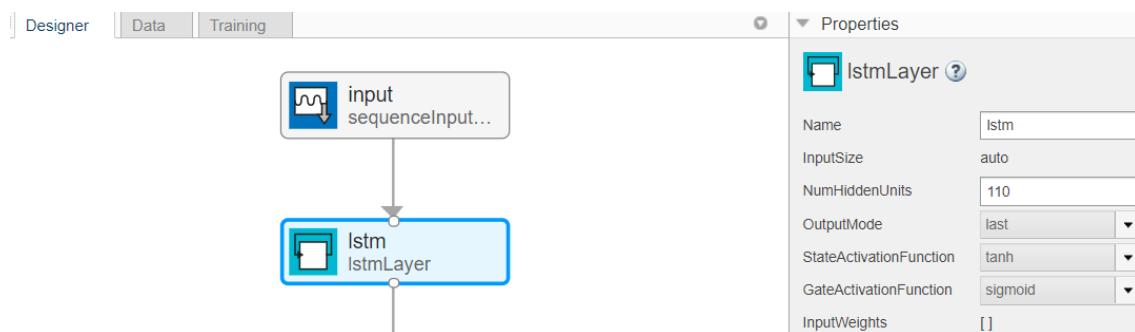


Fig. 103 – Configuración de la capa LSTM

Mediante “Analyze” verificar que la red no tenga errores, ver Fig. 104.

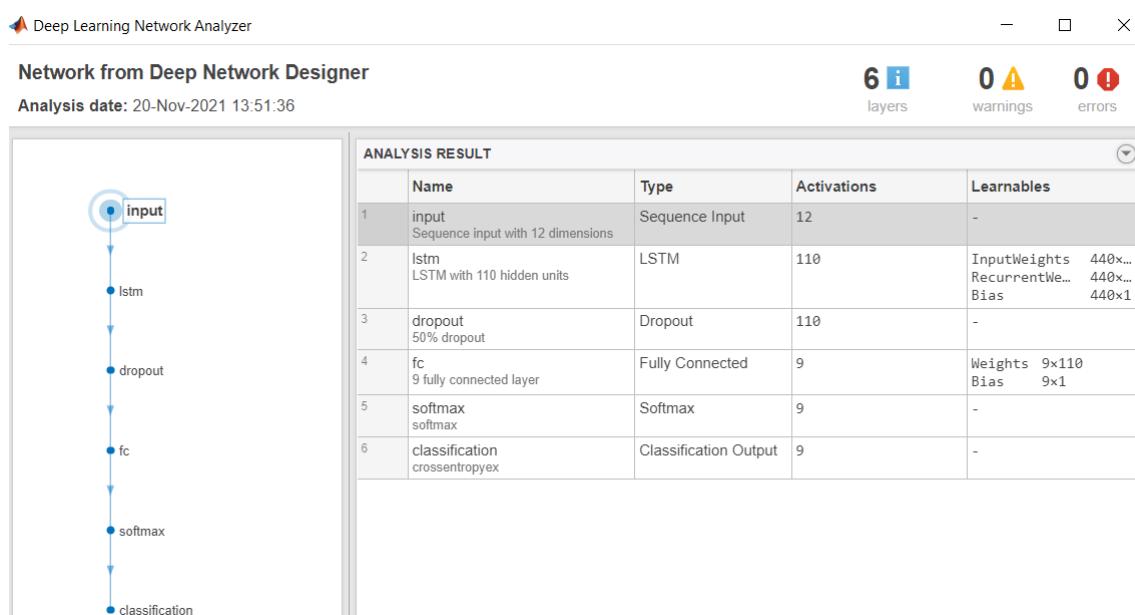


Fig. 104 – Detalles de la red LSTM

Luego exportamos la arquitectura de la red al espacio de trabajo, se debe apretar el comando “Export” y se guarda la red con el nombre layer_1. También se puede generar código para construir la arquitectura de la red seleccionando Export > Generate code.

A continuación, se muestra el programa utilizado

```
% Diseñamos la red
deepNetworkDesigner
```

```
% Cargamos conjunto de datos
```

```
[X_entrenenam,Y_entrenenam] = japaneseVowelsTrainData;
[X_Validacion,Y_Validacion] = japaneseVowelsTestData;
% Mostramos algunos datos
X_entrenenam(1:5)
% Especificamos opciones de entrenamiento y entrenamos la red
% Al tener secuencias cortas, utilizamos 'cpu'
miniBatchSize = 27;
opciones = trainingOptions('adam', 'ExecutionEnvironment', 'cpu', 'MaxEpochs', 120, ...
    'MiniBatchSize', miniBatchSize, 'ValidationData', {X_Validacion, Y_Validacion}, ...
    'GradientThreshold', 2, 'Shuffle', 'every-epoch', 'Verbose', false, 'Plots', 'training-progress'); %
net1 = trainNetwork(X_entrenenam, Y_entrenenam, layers_1, opciones);
% Probamos la red
% Especificar el mismo tamaño de MiniBatchSize que para el entrenamiento.
YPredecida = classify(net1, X_Validacion, 'MiniBatchSize', miniBatchSize)
precision = mean(YPredecida == Y_Validacion)
% Se obtiene precision promedio: 0.9622
```

En la Fig. 105 se muestra el resultado del entrenamiento, donde se observa una alta precisión en la clasificación.

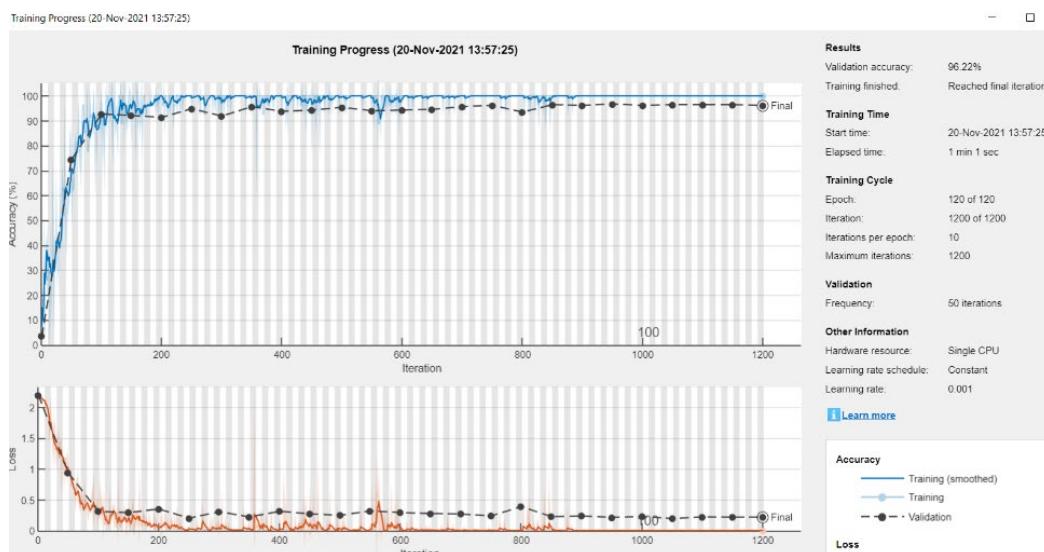


Fig. 105 – Resultados del entrenamiento de la red LSTM para clasificar vocales

Ver descarga del código al final de este capítulo

Ejercicio 6.15

Ejemplo de reconocimiento de voz con spectrogramas, aprendizaje profundo y procesamiento paralelo en Matlab

En este ejemplo se reconocen comandos de voz simples. Se explica el programa en la siguiente página:

<https://la.mathworks.com/help/deeplearning/ug/deep-learning-speech-recognition.html>

Se puede abrir el ejemplo con el siguiente comando en Matlab:

```
openExample('deeplearning_shared/DeepLearningSpeechRecognitionExample')
```

Se utiliza el conjunto de datos: “Google’s Speech Commands”, contiene 65.000 expresiones de un segundo con un total de 30 palabras cortas, pronunciada por miles de personas diferentes. A continuación, se muestra la red profunda utilizada.

```
layers = [ imageInputLayer([ numHops numBands ] )
    convolution2dLayer( 3, numF, 'Padding', 'same' )
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer( 3, 'Stride', 2, 'Padding', 'same' )
    convolution2dLayer( 3, 2*numF, 'Padding', 'same' )
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer( 3, 'Stride', 2, 'Padding', 'same' )
    convolution2dLayer( 3, 4 *numF, 'Padding', 'same' )
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer( 3, 'Stride', 2, 'Padding', 'same' )
    convolution2dLayer( 3, 4*numF, 'Padding', 'same' )
    batchNormalizationLayer
    reluLayer
    convolution2dLayer( 3, 4 *numF, 'Padding', 'same' )
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer( [ timePoolSize, 1] )
    dropoutLayer( dropoutProb )
    fullyConnectedLayer( numClasses )
    softmaxLayer
    weightedClassificationLayer( classWeights ) ];
```

En la Fig. 106 se muestran 3 ejemplos de voces en dominio temporal, y sus respectivos espectrogramas. En la Fig. 107 se muestran la matriz de confusión obtenida. En este ejemplo solo se seleccionaron 10 clases de palabras del conjunto de datos completo.

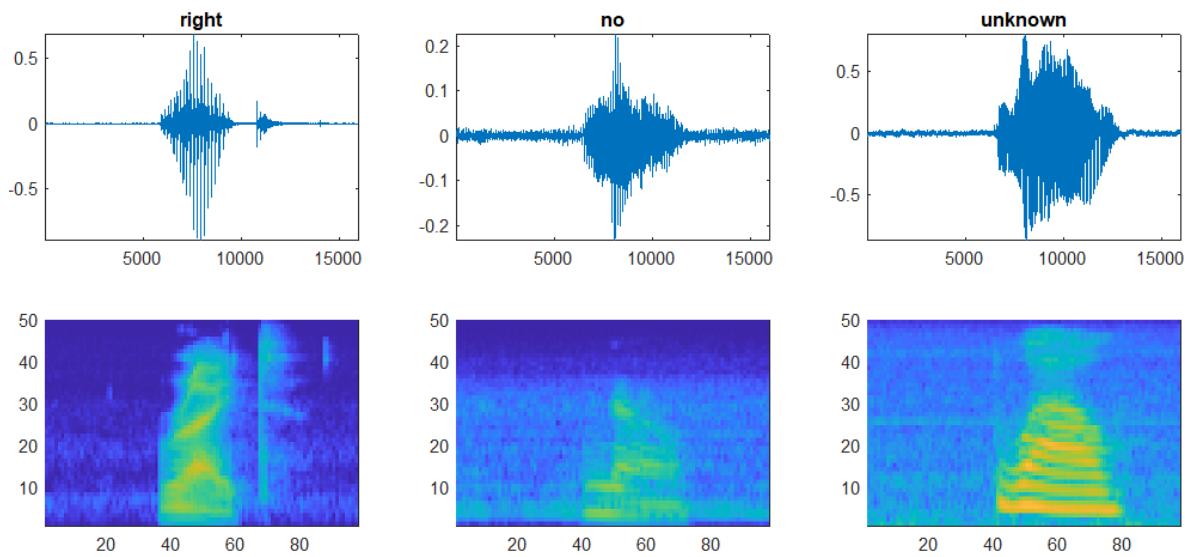


Fig. 106 – Ejemplo de voces y espectrogramas

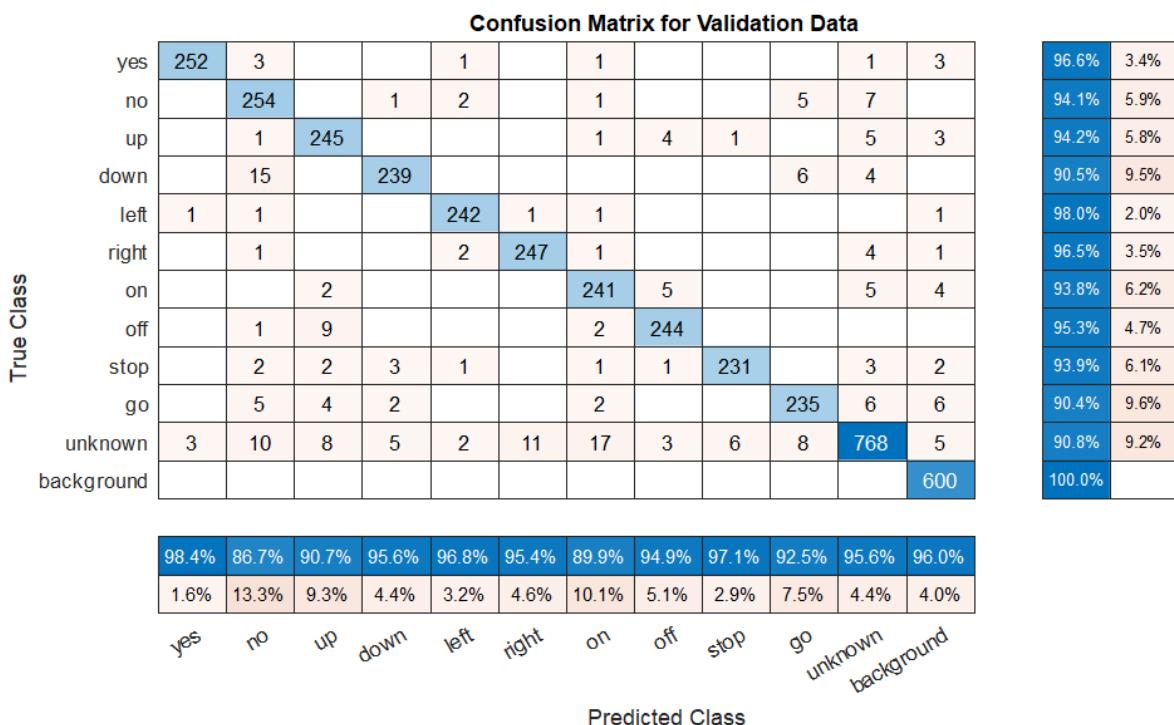


Fig. 107 – Matriz de confusión para reconocimiento de voz



Descarga de los códigos de los ejercicios

Referencias

- Beale, M. H., Hagan, M., & Demuth, H. (2020). Deep Learning Toolbox™ User's Guide. In MathWorks. <https://la.mathworks.com/help/deeplearning/index.html>
- Carreira, J., & Zisserman, A. (2017). Quo Vadis, action recognition? A new model and the kinetics dataset. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* (Vol. 2017-January). <https://doi.org/10.1109/CVPR.2017.502>
- Demuth H, Beale M, Hagan M. (2018). Neural Network Toolbox™ User's Guide Neural network toolbox. MathWorks.
- Hagan, M. T., Demuth, H. B., Beale, M. H., & De Jesus, O. (2014). Neural Network Design 2nd Edition. In *Neural Networks in a Soft computing Framework*.
- Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. In *ACL-HLT 2011 - Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies* (Vol. 1).
- Pytorch, Biblioteca de aprendizaje automático Pytorch, 2021. <https://pytorch.org/>
- Scikit-learn, biblioteca de software de aprendizaje automático para Python, 2021. <https://scikit-learn.org/stable/>
- Tensorflow, Biblioteca de aprendizaje automático Tensorflow, desarrollada por Google, 2021, <https://www.tensorflow.org/>
- The Mathworks, Inc. MATLAB, Version 9.6, 2019. (2019). MATLAB 2019b - MathWorks. In www.Mathworks.Com/Products/Matlab.

Capítulo VII –Redes Neuronales Convolucionales

Se realiza una introducción al aprendizaje profundo con redes neuronales convolucionales (CNN). Estas redes son modernas y evolucionaron mucho los últimos años. Trabajan con matrices de dos dimensiones y son muy utilizadas para trabajar con imágenes ya que se obtienen resultados muy satisfactorios. También se utilizan en aplicaciones sin imágenes. Estas redes generalmente son grandes, contienen muchas capas convolucionales que utilizan filtros de convolución. Se muestran las diferencias entre aprendizaje profundo con CNN y aprendizaje automático. Se presentan herramientas y librerías para trabajar con redes CNN y se describen algunas redes preentrenadas. Por último, se muestran varios ejemplos resueltos de construcción de redes, detección de objetos y clasificación de imágenes

Introducción a las Redes Neuronales Convolucionales (CNN)

En 1988 Yann LeCun introduce las CNN mediante la red convolucional LeNet, esto genera nuevas investigaciones y nuevos métodos en la visión por computadora (Kaplunovich, 2020).

Las CNN son supervisadas, es decir se entrena con entradas y salidas. Estas redes imitan la corteza visual del cerebro humano. Contienen muchas capas ocultas jerarquizadas y especializadas. Las primeras capas son las encargadas de detectar curvas y líneas, luego las capas más profundas reconocen formas más avanzadas como rostros, animales, toda clase de objetos, etc. Debido a que trabajan con matrices en 2 dimensiones, son más efectivas para detectar y clasificar imágenes (Beale, 2020), (Boveiri, 2020).

Las CNN contienen las siguientes capas:

- Convolución: compuesta por filtros convolucionales con núcleos para detectar ciertas características de la imagen.
- ReLU o Unidad lineal rectificada o también conocida como función de rectificación: rectifican las señales, es decir sus salidas siempre son positivas, esta capa de entrenamiento permite un procesamiento más rápido y eficiente.
- Submuestreo o Pooling: para bajar la carga computacional, esta capa reduce el tamaño de la imagen utilizando tasas de muestreo más bajas.

Luego de estas tres capas, las CNN utilizan una red de clasificación para hallar la salida (Chaudhary, 2020), (Gross, 2019).

La Fig. 108 muestra un ejemplo de convolución que reduce 3x3 píxeles a 1 píxel en forma sucesiva.

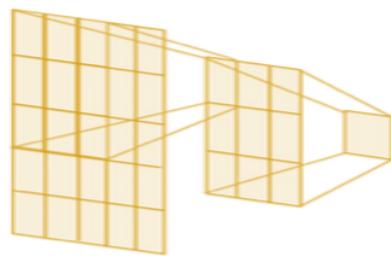


Fig. 108 – Convolución sucesiva de 3x3

Las capas ocultas en una CNN generalmente son capas de convolución, rectificación y pooling (submuestreo con reducción de resolución). En cada capa de convolución, tomamos un filtro pequeño, lo movemos por la imagen y realizamos operaciones de convolución (Basha, 2021). Las operaciones de convolución son las sumas de multiplicación de matrices por elementos entre los valores del filtro y los píxeles de la imagen, ver Ejercicio 7.1.

Los valores del filtro se ajustan en el entrenamiento en forma iterativa. Luego de entrenar la red, estos filtros comienzan a buscar varias características en la imagen. Tomemos el ejemplo de la detección de un rostro mediante una red neuronal convolucional. Las primeras capas de la red buscan características simples como bordes en diferentes orientaciones, etc. A medida que avanzamos a través de la red, las capas comienzan a detectar características más complejas. En las últimas capas, las características detectadas se parecen a diferentes partes de la cara (Demuth, 2018).

Ahora, analizamos las capas de pooling (agrupación o submuestreo). Estas capas reducen la resolución de la imagen. La imagen obtenida contiene pocos valores píxeles, facilitando a la red aprender las características. Se reducen la cantidad de parámetros requeridos y, por lo tanto, esto reduce el cálculo requerido. La agrupación también ayuda a evitar el sobreajuste. Se pueden elegir dos tipos de operaciones en la capa de pooling:

Agrupación máxima: selección del valor máximo

Agrupación promedio: calcula el promedio de todos los valores, se usa muy poco.

En la Fig. 109 se muestra un ejemplo de capa pooled donde se reduce una matriz de 25 x 25 a otra matriz de 4x4.

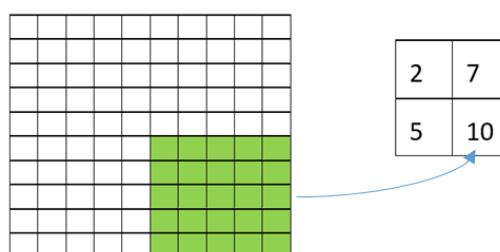


Fig. 109 – Capa Pooled

La Fig. 110 muestra un ejemplo de CNN, en las primeras capas se realiza el aprendizaje de características (en inglés feature learning) y al final se realiza la clasificación.

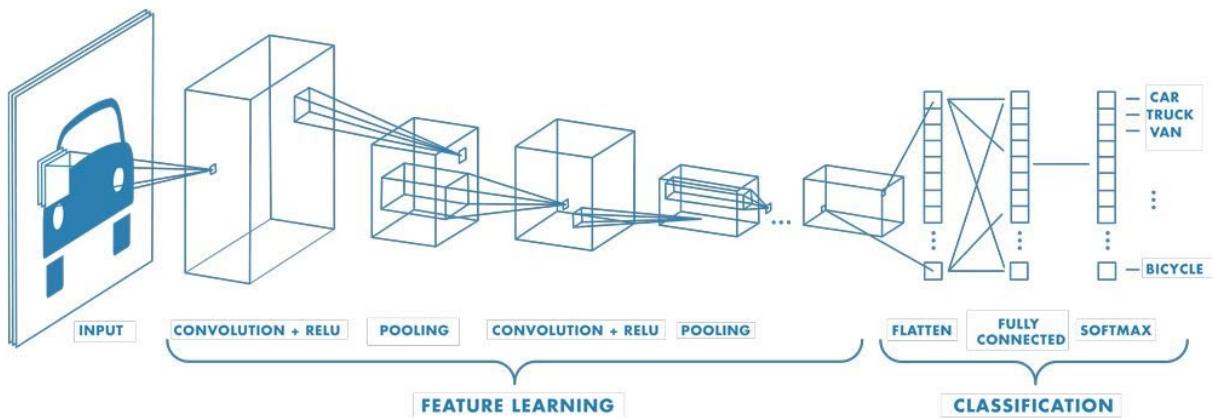


Fig. 110 – Ejemplo de Red Neuronal Convolucional (CNN)

Fuente de la imagen:

<https://la.mathworks.com/discovery/convolutional-neural-network-matlab.html>

En las redes de aprendizaje profundo CNN (convolutional neural network en inglés) la extracción de características de las muestras la realiza la red. Mientras que en el Aprendizaje Automático (en inglés: Machine Learning) las redes requieren una etapa previa para la extracción de características, ver Fig. 111, (Demuth,2018).

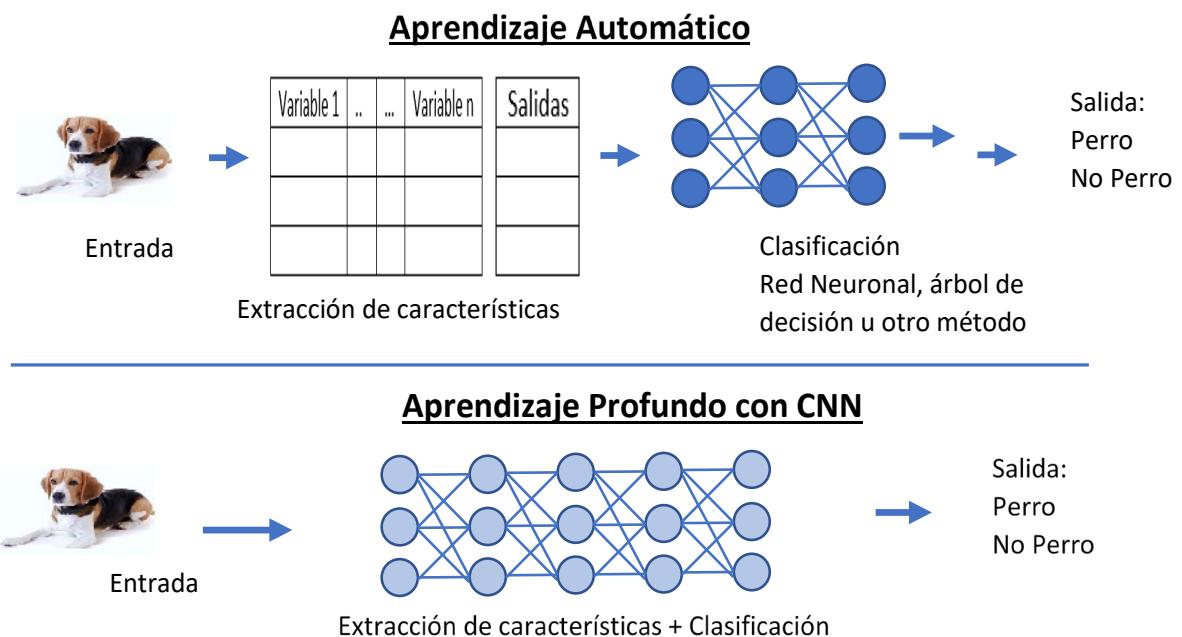


Fig. 111 – Diferencias entre aprendizaje profundo y aprendizaje automático

Herramientas y librerías avanzadas para redes CNN

El entorno Python dispone muchas librerías avanzadas de Inteligencia Artificial, a continuación, se nombran algunas:

- Scikit-learn
<https://scikit-learn.org/stable/>
- Pytorch
<https://pytorch.org/>
- Tensorflow
<https://www.tensorflow.org/>

En Matlab® se puede utilizar la herramienta “Deep Network Designer” para simplificar la construcción de las redes de aprendizaje profundo (The Mathworks, 2019).

<https://la.mathworks.com/>

Las CNN requieren mucha cantidad de imágenes para su correcto funcionamiento. Se pueden utilizar bases de datos públicas, por ejemplo, ImageNet en 2009 llegó a 3.2 millones de imágenes. En el año 2012 AlexNet es la primera red CNN que se entrena con procesadores NVIDIA GeForce 256 GPU (en inglés GPU: Graphics Processing Unit). También aparecen nuevas redes preconfiguradas, mencionamos solamente algunas:

- Alexnet
- Googlenet/Inception
- ResNet
- VGG-19
- Inception

En la Tabla XIII se muestran algunas redes preconfiguradas disponibles en Keras Tensorflow

Respecto a los datos, se puede acceder en forma pública a muchos conjuntos diferentes de datos de entrenamiento con imágenes, enumeramos solo algunos:

- ImageNet ahora dispone de más de 14 millones clasificadas con 20000 etiquetas.
- CIFAR10 contiene 60000 imágenes, donde cada imagen tiene un tamaño 32x32x3 en formato RGB. Posee 10 clases: automóvil, avión, ave, gato, venado, rana, perro, caballo, camión y barco.
- MNIST contiene imágenes con dígitos del 0 al 9, dispone 60000 imágenes de entrenamiento y 10000 de prueba.

Tabla XIII – Ejemplo de algunas redes preconfiguradas disponibles en Keras Tensorflow

Modelo	Tamaño	Precisión para primer resultado	Precisión para primeros 5 resultados	Profundidad	Parámetros
Xception	88 MB	0,790	0,945	126	22.910.480
VGG16	528 MB	0,713	0,901	23	138.357.544
VGG19	549 MB	0,713	0,9	26	143.667.240
ResNet50	98 MB	0,749	0,921	-	25.636.712
ResNet152V2	232 MB	0,78	0,942	-	60.380.648
InceptionV3	92 MB	0,779	0,937	159	23.851.784
MobileNetV2	14 MB	0,713	0,901	88	3.538.984
DenseNet201	80 MB	0,773	0,936	169	20.242.984
EfficientNetB7	256 MB	-	-	-	66.658.687

Ejercicios

Ejercicio 7.1

Ejemplo de convolución continua. Resolución analítica.

Se pide resolver la siguiente convolución de la Fig. 112.

Datos originales

1	1	0	0	0
0	1	1	0	0
0	0	1	1	1
0	0	0	1	0
0	1	1	0	0

Máscara de convolución

1	0	0
0	1	0
1	0	1

Máscara de convolución (agregamos x)

x1	x0	x0
x0	x1	x0
x1	x0	x1

Fig. 112 – Enunciado Convolución analítica

Resolución

Calculamos convolución en la posición 1

1	x1	1	x0	0	x0	0	0
0	x0	1	x1	1	x0	0	0
0	x1	0	x0	1	x1	1	1
0	0	0	1	0	1	0	0
0	1	1	0	0	0	0	0

$$\begin{aligned} 1x1 + 1x0 + 0x0 + \\ 0x0 + 1x1 + 1x0 + \\ 0x1 + 0x0 + 1x1 \end{aligned}$$

Resultados

3		

Posición 2

1	1	x1	0	x0	0	x0	0
0	1	x0	1	x1	0	x0	0
0	0	x1	1	x0	1	x1	1
0	0	0	1	0	1	0	0
0	1	1	0	0	0	0	0

Posición 3

1	1	0	x1	0	x0	0	x0
0	1	1	x0	0	x1	0	x0
0	0	1	x1	1	x0	1	x1
0	0	0	1	0	1	0	0
0	1	1	0	0	0	0	0

Posición 9

1	1	0	0	0	0	
0	1	1	0	0	0	
0	0	1	x1	1	x0	1
0	0	0	x0	1	x1	x0
0	1	1	0	0	0	0

Resultados:

3	3	2
0	3	2
1	1	3

Fig. 113 – Ejemplo de Convolución analítica

Ejercicio 7.2

Ejemplo simple de red neuronal “GoogleNet” Recurrente (RNN) en Matlab®, para identificación de imágenes.

Analizar el funcionamiento de la red Googlenet mediante el siguiente programa. Se puede usar imágenes simples descargadas de internet para clasificar. O también se puede usar la siguiente imagen de Matlab para clasificar:

```
I1 = imread('peppers.png');
```

Programa

```
%% Clasificación de Imágenes mediante Red Neuronal GoogLeNet en Matlab®.
% Utilizamos red neuronal convolucional de aprendizaje profundo GoogLeNet.
% Esta red esta entrenada con más de 1 millón de imágenes. Clasifica
% imágenes en 1000 categorías (taza, teclado, café, lápiz, muchos animales, etc.)
clc ; clear all ; close all; nnet.guis.closeAllViews()
```

```
% Cargamos la Red entrenada previamente con el comando googlenet
net1 = googlenet;
% Analizamos la estructura de la red
% analyzeNetwork funciona en Matlab 2018a. Instalar paquete: network_analyzer.mlpkginstall
analyzeNetwork(net1);
% Mostramos el tamaño que tiene la imagen.
size1_input = net1.Layers(1).InputSize
% Mostramos 15 clases en forma aleatoria de un total de 1000.
nombre_clases = net1.Layers(end).ClassNames;
cantidad_clases = numel(nombre_clases);
disp(nombre_clases(randperm(cantidad_clases,15)));
% Leemos una imagen para Clasificar y modificamos su tamaño.
I1 = imread('imagen_manzanas.jpg');
figure ; imshow(I1)
size(I1)
% Tamaño: 384-by-512 pixeles, 3 colores (RGB). Cambiamos tamaño con imresize
I1 = imresize(I1, size1_input(1:2));
% Clasificamos la imagen
[etiqueta1 ,scores1] = classify(net1,I1);
etiqueta1
% Mostramos probabilidad de predicción y la imagen con la etiqueta.
figure ; imshow(I1)
titulo = [string(etiqueta1) + ", " + ...
    num2str(100*scores1(nombre_clases == etiqueta1),3) + "%"];
title(titulo)
% Buscamos los 7 resultados con mayor probabilidad.
% La red clasificó la imagen como: 'bell pepper' con alta probabilidad.
 [~,idx1] = sort(scores1, 'descend');
idx1 = idx1(7:-1:1);
nombre_clases_mayor_prob = net1.Layers(end).ClassNames(idx1);
scores_Top1 = scores1(idx1);
% Graficamos
figure ; barh(scores_Top1) ; xlim([0 1])
title('7 Predicciones con mayor probabilidad')
xlabel('Probabilidad') ; yticklabels(nombre_clases_mayor_prob)
%% Clasificamos 4 imágenes distintas
figure ;
lista_imagenes = {'imagen_siameses.jpg', 'imagen_perros.jpg', ...
    'imagen_pimiento_morron.jpg', 'imagen_auto.jpg'};
for i=1:4
    I1 = imread( lista_imagenes{i} );
    I1 = imresize(I1, size1_input(1:2));
    [etiqueta1 ,scores1] = classify(net1,I1);
    subplot(2,2,i); imshow(I1)
    titulo = [string(etiqueta1) + ", " + ...
        num2str(100*scores1(nombre_clases == etiqueta1),3) + "%"];
    title(titulo)
end
```

Resultados

Se muestran solo algunos nombres traducidos de clases: Ponche de huevo, castillo, apósito adhesivo, bolsa de dormir, cinturón de seguridad, naranja, vaca, pimientos morrones.

La Fig. 114 muestra solo las primeras capas de la red googlenet, con la función *analyzeNetwork* se muestra en forma detallada la estructura completa.

En la Fig. 115 y Fig. 116 se muestran los resultados de la clasificación de manzanas donde se asigna probabilidad de 92,7 % para manzanas verdes. En la Fig. 117 se muestran los resultados de la clasificación de 4 imágenes distintas.

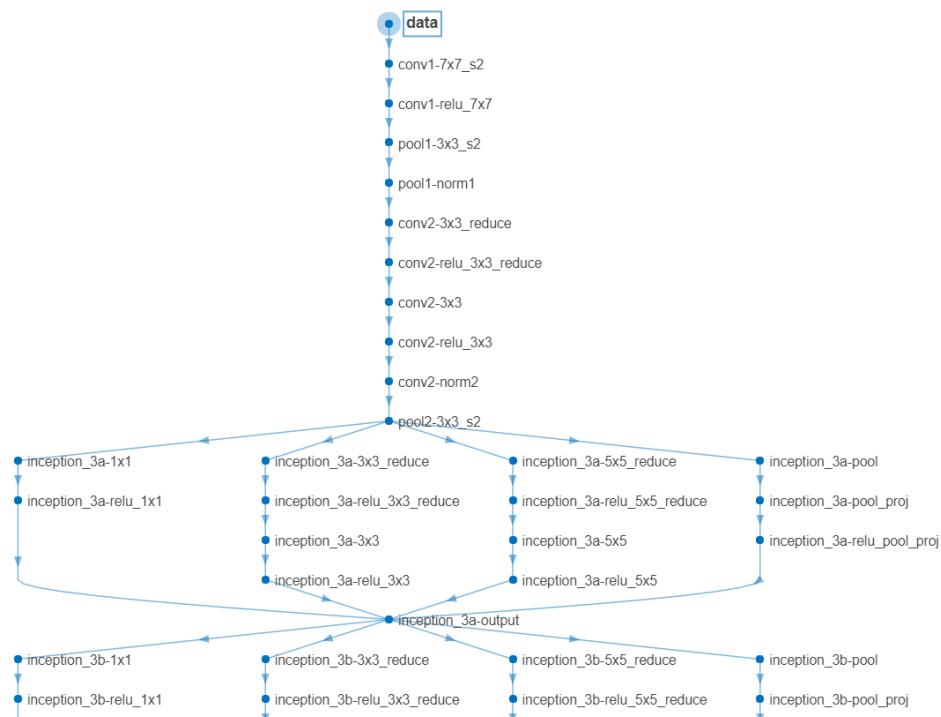


Fig. 114 – Estructura de las primeras capas de la red Googlenet, contiene 144 capas.

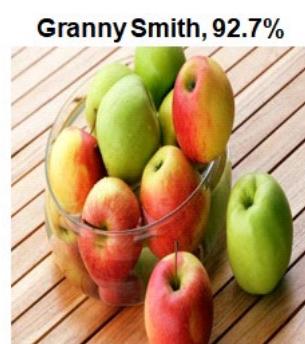


Fig. 115 – Clasificación de manzanas

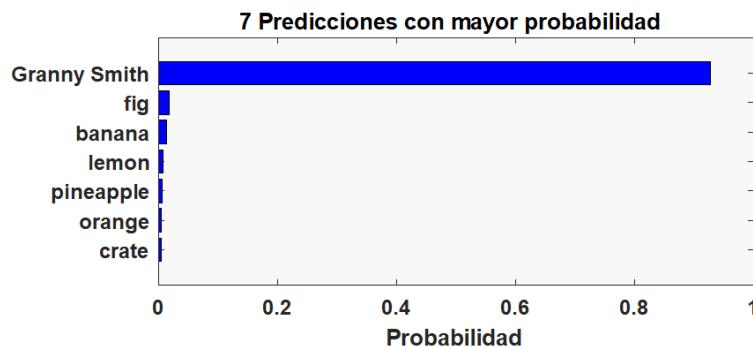


Fig. 116 – Resultados de la predicción

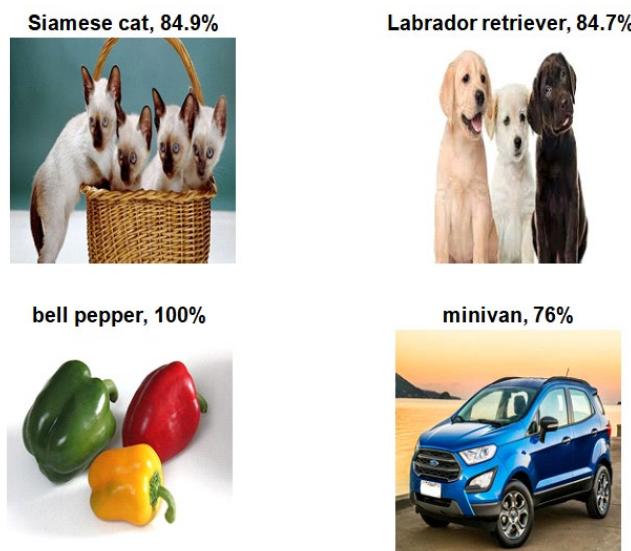


Fig. 117 – Resultados de la clasificación de 4 imágenes y sus probabilidades.

Ejercicio 7.3

Ejemplo de carga y visualización de redes CNN preentrenadas con herramienta `deepNetworkDesigner` de Matlab®

En la versión 2018b de Matlab® se incorpora una herramienta gráfica para facilitar el diseño de redes de aprendizaje profundo. En este ejemplo se muestra como cargar, visualizar y exportar redes preentrenadas mediante la herramienta `deepNetworkDesigner`. A modo de ejemplo se pueden cargar las siguientes redes: SqueezeNet, GoogleNet, ResNet-50, EfficientNet-b0, DarkNet-53, etc. Ver Fig. 118.



Fig. 118 – Ejemplo de redes preentrenadas

En la consola de Matlab® se debe escribir el siguiente comando:

```
>> deepNetworkDesigner
```

Se abre una página de inicio, ver Fig. 119, seleccionar y cargar la red preentrenada GoogleNet. Una vez cargada la red se puede visualizar y analizar cada una de las capas de la red. Luego se puede exportar la misma, con el comando “Export”, ver Fig. 120. En la Fig. 121 se muestra una ampliación de la vista de la red.

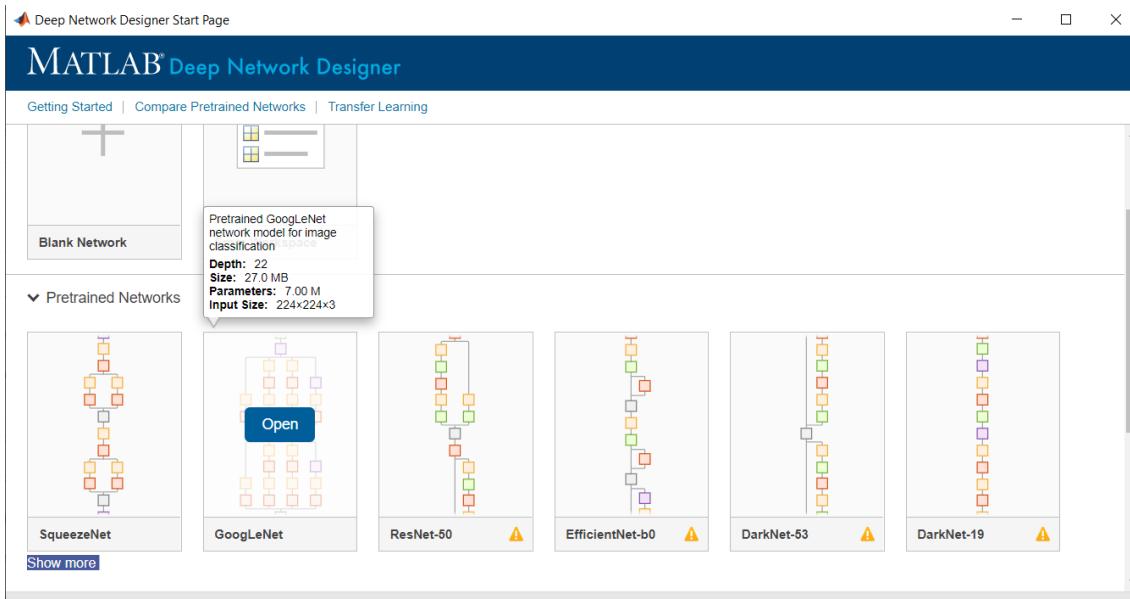


Fig. 119 – Página de inicio de la herramienta deepNetworkDesigner

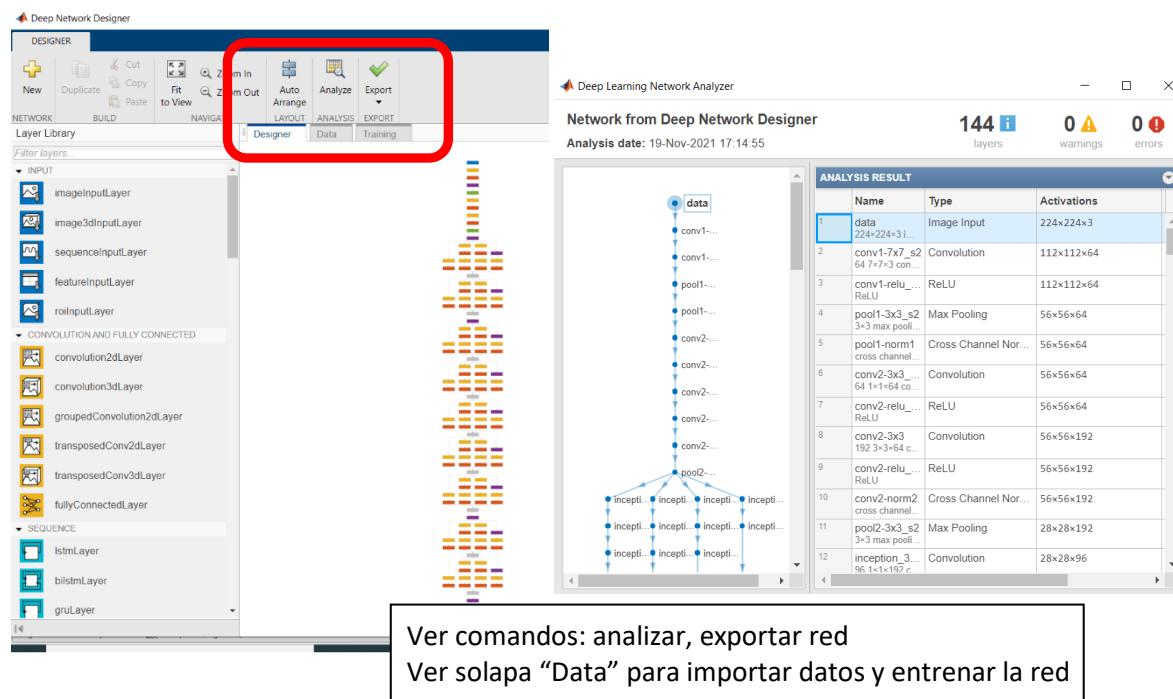


Fig. 120 – Vista de la red GoogleNet. Izq.) esquema de la red con 144 capas, con recuadro en rojo de comandos principales, der.) detalles de cada capa de la red

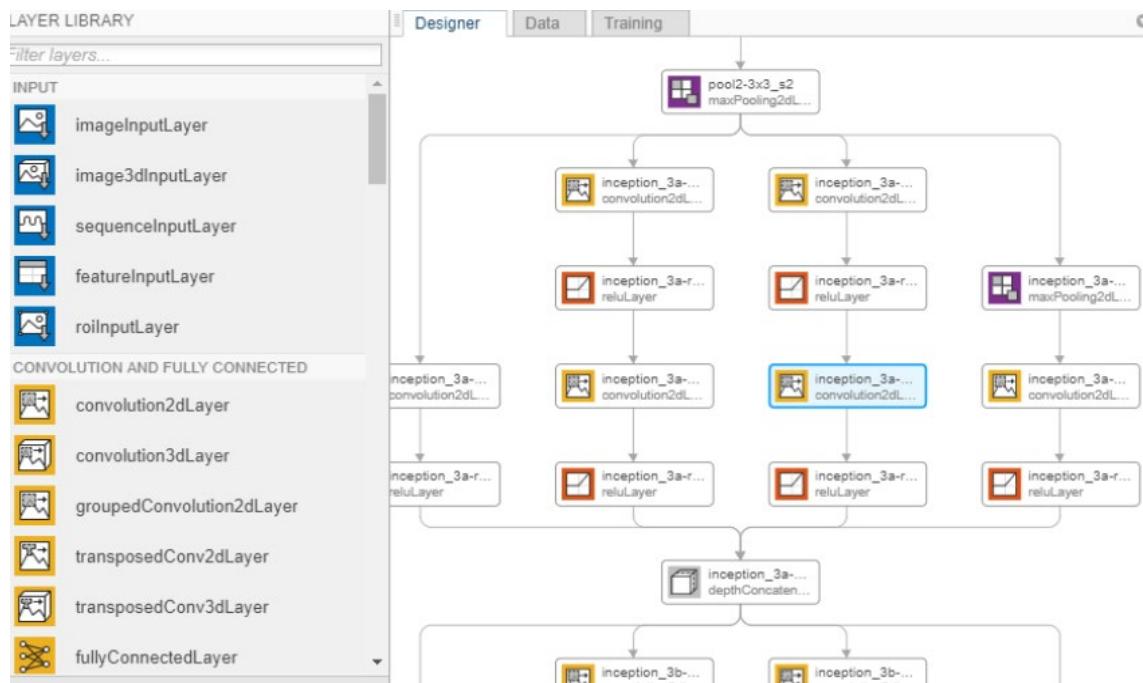


Fig. 121 – Vista ampliada de la red GoogleNet mediante herramienta deepNetworkDesigner de Matlab®.

Ejercicio 7.4

Mediante la herramienta deepNetworkDesigner de Matlab editar la red CNN SqueezeNet preentrenada, importar datos, entrenar y clasificar

El entrenamiento nuevo de una red preentrenada se puede realizar con pocos datos, resulta más fácil y rápido que entrenar una red desde cero. Hay que tener en cuenta de cambiar el número de clases para que coincida con el nuevo conjunto de datos. Mediante el siguiente comando cargamos la red CNN SqueezeNet preentrenada.

```
>> deepNetworkDesigner(squeezeNet)
```

Hay que reemplazar la última capa capaz de entrenar y la capa de clasificación final. Para SqueezeNet, modificar la capa llamada 'conv10', ver Fig. 122.

Arrastrar una nueva capa convolution2dLayer al esquema. Establecer propiedad FilterSize en 1,1 y la propiedad NumFilters con en el nuevo número de clases (5). Luego cambiar las tasas de aprendizaje en esta nueva capa aumentando los valores de WeightLearnRateFactor y BiasLearnRateFactor, utilizar 10. Eliminar la última convolution2dLayer y conectar la nueva capa en su lugar, ver Fig. 122.

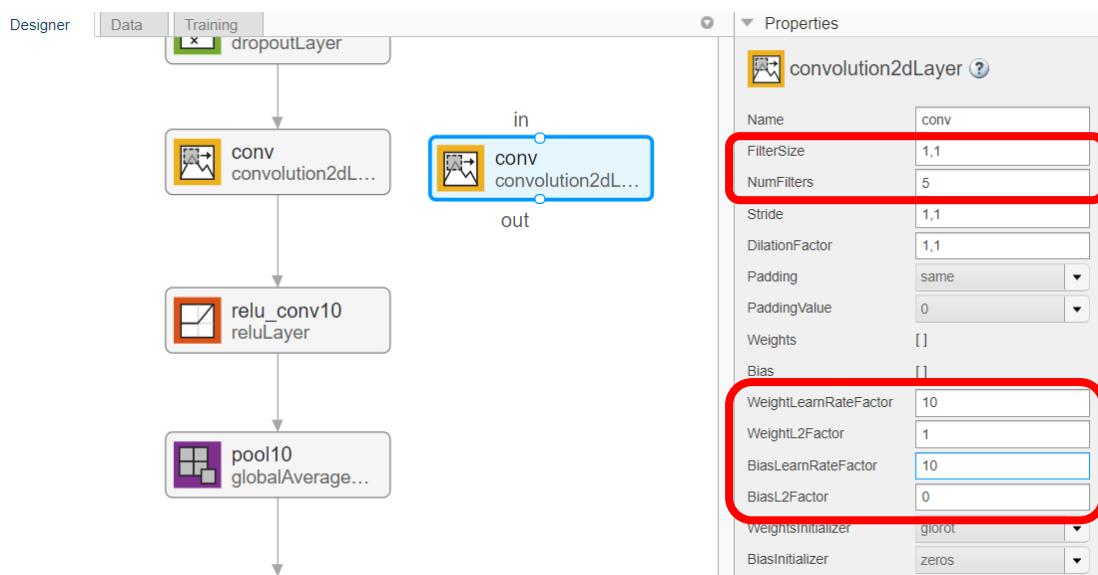


Fig. 122 – Detalles de sustitución de capa convolucional

A continuación, eliminar la capa de salida de clasificación, arrastrar una nueva capa de clasificación al esquema y conectarla en su lugar. La configuración predeterminada en la capa de salida es que la red asigne el número de clases en el entrenamiento, ver Fig. 123.

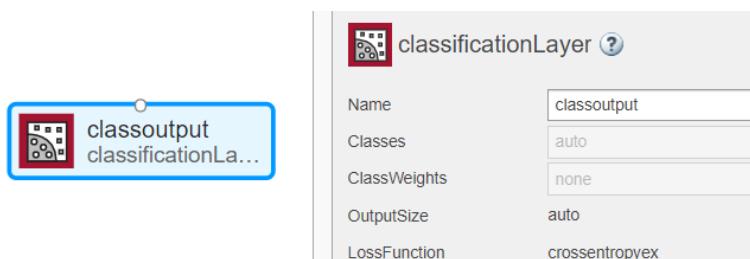


Fig. 123 – Detalles de la capa de clasificación

Verificar la red haciendo clic en “Analyze”, si no se indican errores, la red está lista para entrenar con nuevos datos.

A continuación, se indican los pasos para importar datos. En la línea de comandos escribir el siguiente código para descomprimir un pequeño conjunto de datos de prueba de Matlab.

```
>> unzip("MerchData.zip")
```

Luego importar los datos, ver Fig. 124. Se puede aumentar la cantidad de datos de entrenamiento aplicando un aumento aleatorio a los mismos. El aumento también permite entrenar redes para que sean invariantes a las distorsiones en las imágenes. Por ejemplo, se puede agregar rotaciones aleatorias a las imágenes de entrada para que la red sea invariable a la presencia de rotación en los datos de entrada. En este ejemplo, aplicar una rotación aleatoria del rango [-90,90] grados y una reflexión aleatoria en el eje y.

Se elige un 30 % de datos de validación y se importan los datos. Una vez importados se pueden visualizar y observar las diferentes clases.

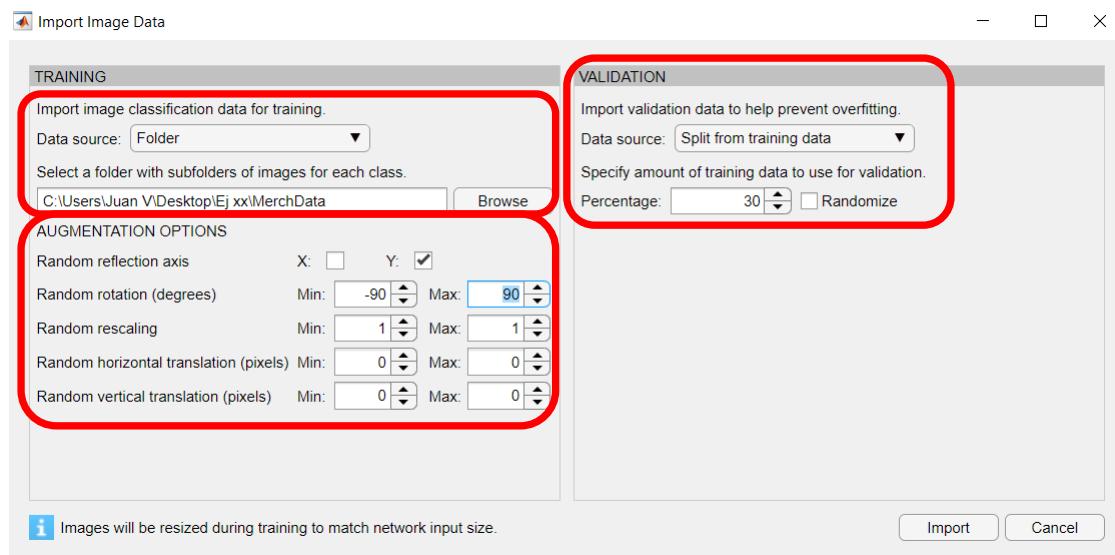


Fig. 124 – Detalles para importar datos

Entrenamiento de la red preentrenada

Modificar las opciones de entrenamiento con los parámetros de la Fig. 125 y entrenar la red. En la Fig. 126 se muestran los resultados donde se observa alta precisión en la clasificación.

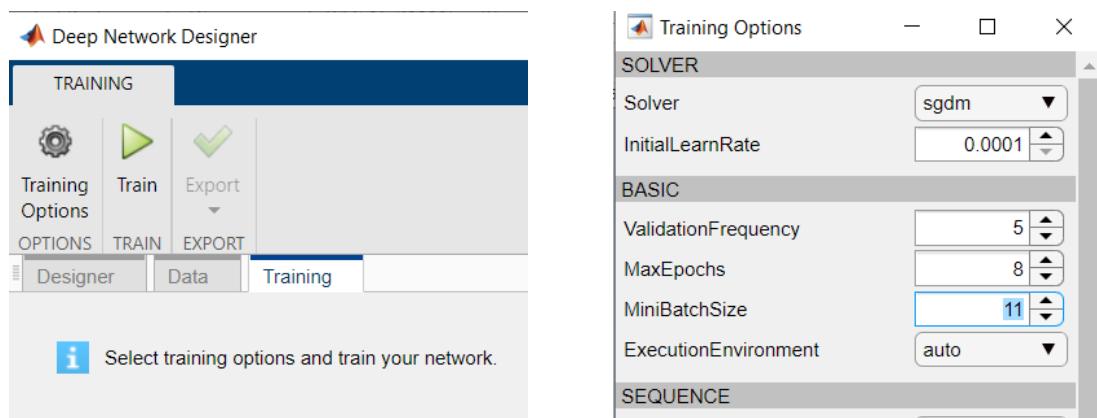


Fig. 125 – Izq.) Pantalla de entrenamiento, der.) opciones de entrenamiento

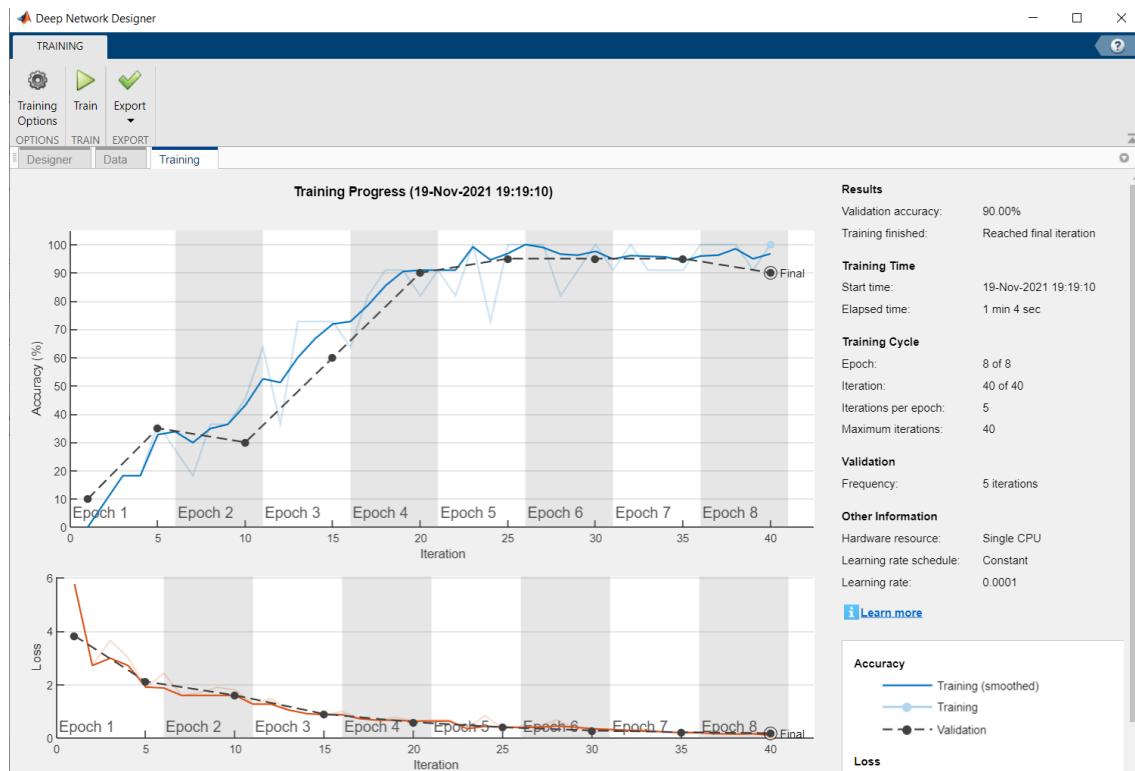


Fig. 126 – Resultados obtenidos del entrenamiento nuevo de la red CNN

Exportar resultados

Para exportar la arquitectura de red entrenada con los nuevos datos, en la pestaña “Training”, seleccionar: “Export”. Se exporta la red entrenada como la variable `trainingNetwork_1` y la información de entrenamiento como la variable `trainInfoStruct_1`.

Generar código Matlab

Export > Generate Code for Training

Clasificar una imagen nueva

Para clasificar una imagen, primero hay que cambiar el tamaño de la imagen de prueba, al tamaño de entrada de la red. Este tamaño se puede ver en la primera capa, ver Fig. 127. En este ejemplo es de 227x227



Fig. 127 –Detalles para conocer el tamaño de entrada de la red.

A continuación, se muestra el código para clasificar una imagen y en la Fig. 128 se muestran los resultados.

```
% Código para importar la imagen de prueba, modificar su tamaño y clasificar.
load matlab_workspace % Cargamos la red
I1 = imread( "MerchDataTest.jpg" );
I1 = imresize( I1, [ 227 227 ] );
[ YPredec1 ,probab1 ] = classify( trainedNetwork_1,I1 ) ;
imshow(I1)
etiqueta1 = YPredec1;
title(string(etiqueta1) + ", " + num2str(max(probab1)*100,3) + "%");
```



Fig. 128 – Resultados de clasificación con red CNN SqueezeNet modificada

Ejercicio 7.5

Diseño de redes CNN para clasificación de números en Lenguaje Matlab®. con conjunto de datos *Digits Dataset*.

Se muestra un ejemplo para detectar imágenes con números (ver Fig. 129) utilizando una red CNN.

En el programa se obtiene una precisión alta mayor al 98%.

Se pide analizar el funcionamiento del siguiente programa. Luego se pide reducir la estructura de la red CNN y escribir conclusiones.

```

%% Ejemplo de Red CNN para clasificar números
% Importamos los datos de Digits Dataset
Path_imagenes1 = fullfile(matlabroot,'toolbox','nnet','nnndemos','nndatasets','DigitDataset');
conjunto_imagenes = imageDatastore(Path_imagenes1, 'IncludeSubfolders',true,...,
    'LabelSource','foldernames');
% Graficamos algunas imágenes
figure;
data1 = randperm(10000,12);
for i = 1:12
    subplot(3,4,i);
    imshow(conjunto_imagenes.Files{data1(i)});
end
cuenta_Etiquetas = countEachLabel(conjunto_imagenes)
imagen1 = readimage(conjunto_imagenes,1);
size(imagen1)
% Armamos conjunto de entrenamiento y de validación
num_archivos_entr = 600;
[conjunto_imagEntrenamiento,conjunto_imagValidacion] = ...
    splitEachLabel(conjunto_imagenes, num_archivos_entr,'randomize');
%% Definimos la arquitectura de la red CNN
capas = [
    imageInputLayer([28 28 1]) % entradas
    convolution2dLayer(3,8,'Padding',1) ; % capa convolucional
    batchNormalizationLayer; reluLayer ; % capas batch y relu
    maxPooling2dLayer(2,'Stride',2) ; % Pooling
    convolution2dLayer(3,16,'Padding',1) ; % capa convolucional
    batchNormalizationLayer; reluLayer ; % batch y relu
    % Se pueden agregar estas capas:
    % maxPooling2dLayer(2,'Stride', 2) ; % Pooling
    % convolution2dLayer(3,32,'Padding', 1) ; % capa convolucional
    % batchNormalizationLayer; reluLayer ; % batch y relu ;
    fullyConnectedLayer(10) ;
    softmaxLayer ;
    classificationLayer ;]; % Capa de clasificación
% Opciones
opciones = trainingOptions('sgdm', 'MaxEpochs',4, 'ValidationData',conjunto_imagValidacion, ...
    'ValidationFrequency',35, 'Verbose',true, 'Plots','training-progress');
%% Entrenamos la red
net1 = trainNetwork(conjunto_imagEntrenamiento,capas,opciones);
%% Clasificamos datos y probamos la red
SalidaPredecida = classify(net1,conjunto_imagValidacion);
SalidaValidacion = conjunto_imagValidacion.Labels;
precision = sum(SalidaPredecida == SalidaValidacion)/numel(SalidaValidacion)
% Clasificamos solo 1 imagen
figure; imshow(imagen1)
resultado1 = classify(net1, imagen1)
%% Analizamos la estructura de la red

```

```
% analyzeNetwork funciona en Matlab 2018a.
% Se debe instalar la siguiente librería: network_analyzer.mlpkginstall
analyzeNetwork(net1) ; % se puede utilizar: analyzeNetwork(capas)
n_capa = 2; nombre = net1.Layers(n_capa).Name ; %nombre = 'conv_1';
channels = 1:8;
I = deepDreamImage(net1,nombre,channels,'PyramidLevels',1);
figure; % Graficamos las funciones de la red
for i = 1:8
    subplot(2,4,i); imshow(I(:,:,i))
end
%%%%%
% Agregado clasificación de Patentes
s1 = size(imagen1)
I1 = imread('imagen patente recortada.jpg');
I2 = imresize(I1, [28 28]) ; I3 = rgb2gray( I2 );
figure; imshow(I3) ; s3 = size(I3)
resultado2 = classify(net1, I3)
% Agregado clasificación de números manuscritos
I1 = imread('Imagen_8.jpg');
I2 = imresize(I1, [28 28]) ; I3 = rgb2gray( I2 );
I3 = imcomplement(I3) ; %Invertimos los colores
figure; imshow(I3); s3 = size(I3)
resultado3 = classify(net1, I3)
```

En las Fig. 129, Fig. 130, Fig. 131 y Fig. 132 se muestran los resultados del programa con el subconjunto de datos aleatorio, entrenamiento de la red, funciones de la red en la capa convolucional 2 y la arquitectura de la red respectivamente.

En las Fig. 133 y Fig. 134 se muestran las imágenes utilizadas para reconocimiento de patentes y reconocimiento números manuscritos. Se identifican correctamente.

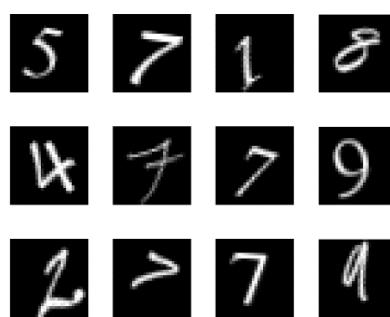


Fig. 129 – Subconjunto de datos aleatorios.

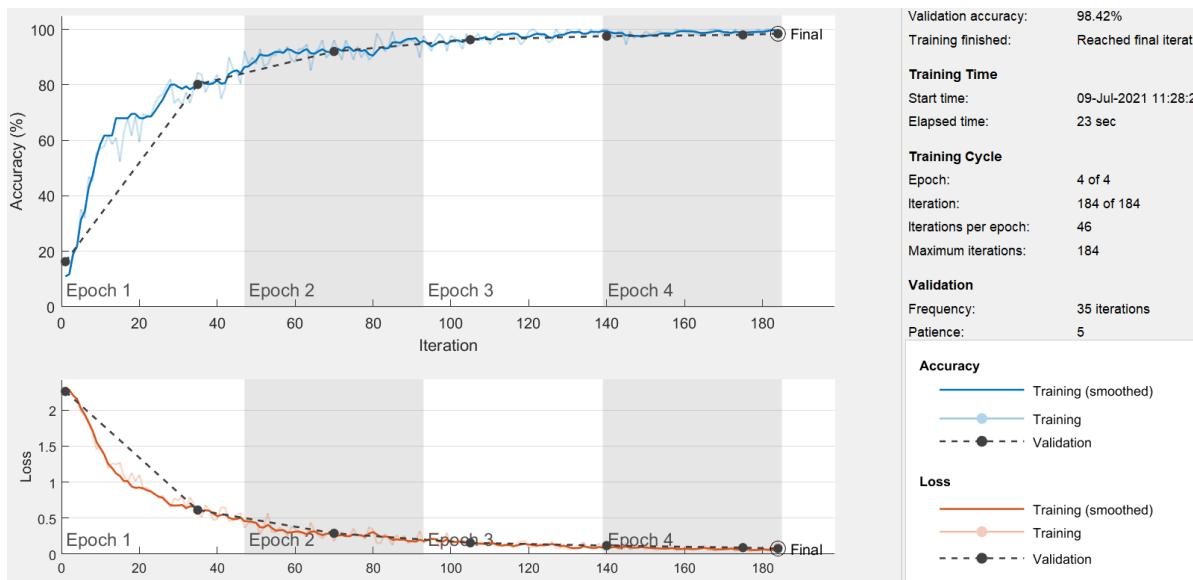


Fig. 130 – Entrenamiento de la red CNN

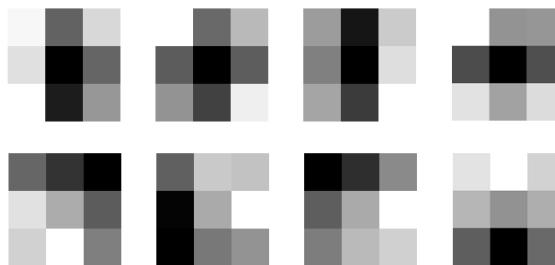


Fig. 131 – Funciones de la red en la capa 2: 'conv_1'

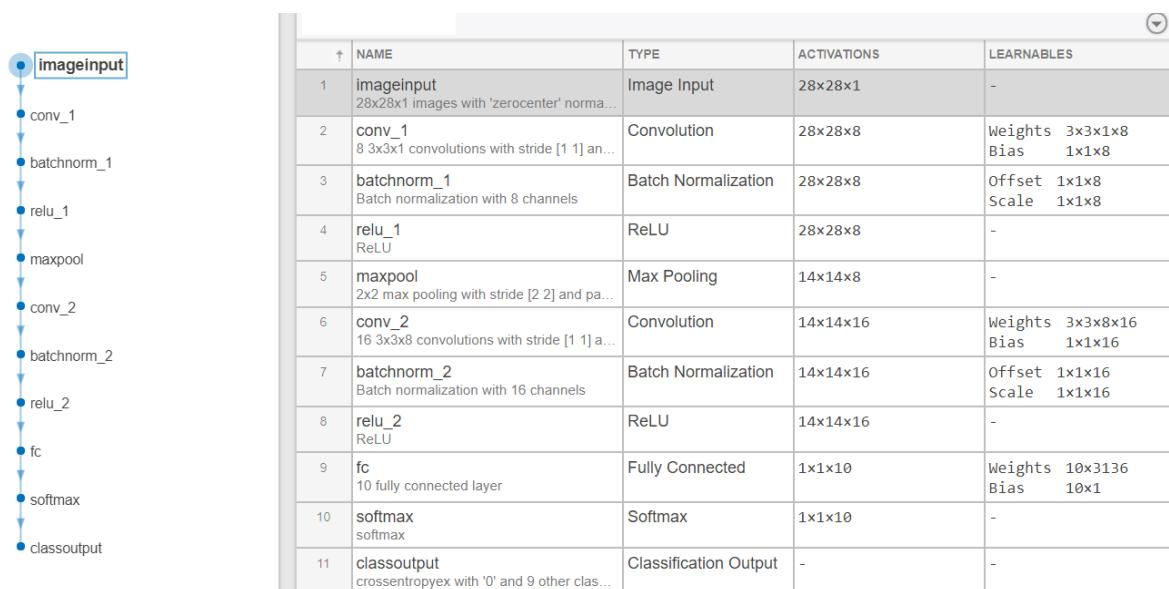


Fig. 132 – Arquitectura de la red CNN, contiene 11 capas.



Fig. 133 – Imágenes para reconocimiento de patentes. Izq.) patente completa, der.) patente recortada.



Fig. 134 – Imagen utilizada para reconocimiento de números manuscritos

Ejercicio 7.6

Clasificación imágenes webcam con red Alexnet. Lenguaje Matlab®.

En este ejemplo mediante la webcam de la PC se clasifican imágenes mediante la red Alexnet

Se pide analizar el siguiente código en Matlab.

```
% Ejemplo de red CNN Alexnet
% Ver mensajes de Matlab: hay que instalar USB camera y AlexNet
mi_camara_web = webcam; % Conectamos la cámara
net1 = alexnet; % Cargamos la red neuronal pre-entrenada Alexnet
for i=1:16
    imag1 = snapshot(mi_camara_web); % Capturamos una imagen
    image( imag1 ); % Mostramos imagen
    imag1 = imresize( imag1,[227 227]); % Cambiamos al tamaño adecuado de la red alexnet
    salida1 = classify( net1, imag1 ); % Clasificamos
    title( char(salida1) ); % Mostramos resultado
    salida1
    drawnow ; pause(0.5) % pausa de duración 0,5 seg.
end
clear mi_camara_web
size( imag1 )
I1 = imread('peppers.png');
% También se puede tomar imagen de internet
%I1 = imread('imagen_manzanas.jpg');
I1 = imresize(I1, [227 227]);
% Se clasifican correctamente las muestras
salida2 = classify( net1, I1 )
% Analizamos la estructura de la red
% analyzeNetwork funciona en Matlab 2018a. Instalar paquete: network_analyzer.mlpkginstall
analyzeNetwork(net1);
```

En la Fig. 135 se muestra la estructura de la red Alexnet.



Fig. 135 – Estructura de la red Alexnet, contiene 25 capas.

Ejercicio 7.7

Ejemplo de clasificación de imágenes con red CNN y conjunto de datos CIFAR10. Lenguaje Python con librerías Pytorch.

CIFAR10 es una colección de datos disponible en internet que contiene 60000 imágenes, separadas en 10 clases. De las cuales 50000 imágenes se utilizan para entrenamiento y las restantes 10000 para prueba.

Las librerías Torch se importan en Python con los siguientes comandos:

```
import torch
import torchvision
```

La red convolucional (CNN) se define por capas mediante una clase. Se muestra un ejemplo de definición de red para clasificar datos CIFAR10.

```
import torch.nn as nn1
import torch.nn.functional as F
class Net1(nn1.Module):
    def __init__(self): # inicio
        super().__init__()
        self.conv1 = nn1.Conv2d( 3, 6, 5 ) ## primera capa convolucional
        self.pool = nn1.MaxPool2d( 2, 2 ) ##
        self.conv2 = nn1.Conv2d( 6, 16, 5 ) ## segunda capa convolucional
        self.fc1 = nn1.Linear( 16 * 5 * 5, 120 ) ##
        self.fc2 = nn1.Linear( 120 , 84 ) ##
        self.fc3 = nn1.Linear( 84 , 10 ) ##
    def forward(self, y): #
        y = self.pool(F.relu(self.conv1(y))) #
        y = self.pool(F.relu(self.conv2(y))) #
        y = torch.flatten(y, 1) # flatten todas menos batch #
        y = F.relu(self.fc1(y)) #
        y = F.relu(self.fc2(y)) #
        y = self.fc3(y) #
        return y
net = Net1()
```

Se muestran algunos ejemplos de imágenes del conjunto de datos públicos CIFAR10, ver Fig. 136.

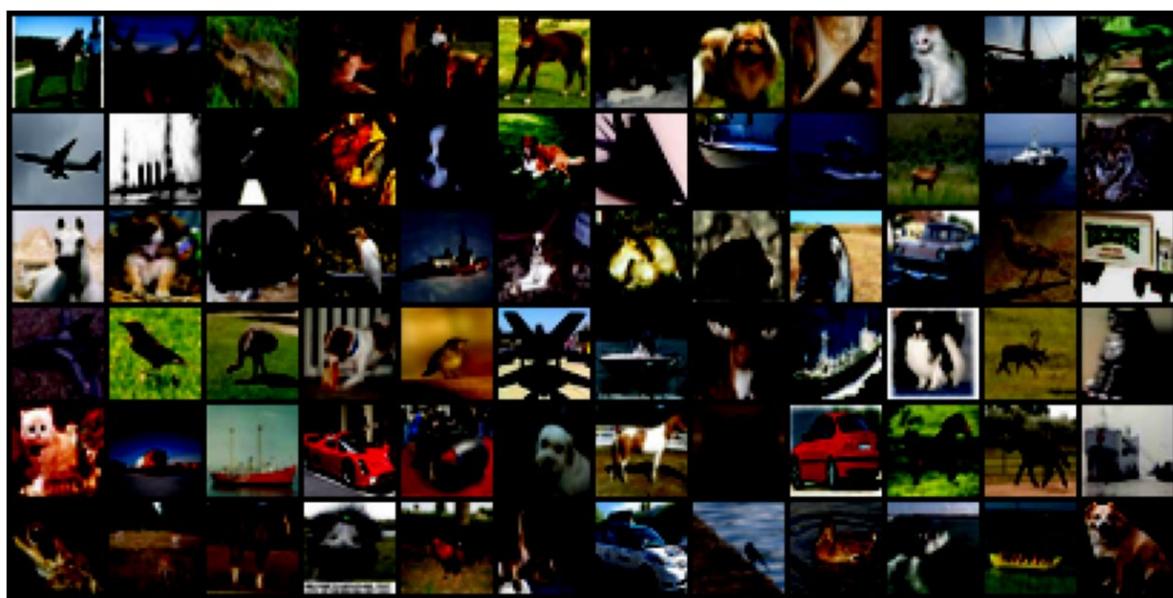


Fig. 136 – Ejemplo de datos del conjunto de imágenes CIFAR10

Resultados

A modo de ejemplo se muestran algunas posibles respuestas al clasificar 4 imágenes:

Caballo gato gato automóvil

Ver descarga del código al final de este capítulo.

Ejercicio 7.8

Clasificación de imágenes con red CNN y conjunto de imágenes propias. Lenguaje Python con librerías Tensorflow.

El siguiente ejemplo construye una red propia CNN. Luego entrena la red con imágenes separadas en carpetas, donde cada carpeta corresponde a la clase correspondiente a la imagen. Luego clasifica una imagen de prueba.

```
from keras.models import Sequential ## Importamos librerías de keras para CNN #
from keras.layers import MaxPooling2D # Capa pooling
from keras.layers import Conv2D # Capa convolucional
from keras.layers import Flatten # Capa Flatten
from keras.layers import Dense # Capa
from keras.preprocessing import image # imágenes
import numpy as np #
from keras.preprocessing.image import ImageDataGenerator ##
# Inicializamos CNN
modelo1 = Sequential()
# Construimos la red CNN
```

```

# Paso 1 convolución 1 y Pooling, paso 2 convolución y Pooling
# Paso 3 Flatten y paso 4 Full connection
modelo1.add( Conv2D(32, ( 3, 3 ), input_shape = (96, 96, 3), activation = 'relu'))
modelo1.add( MaxPooling2D(pool_size = (2, 2)))
modelo1.add( Conv2D(32, (3, 3), activation = 'relu'))
modelo1.add( MaxPooling2D(pool_size = (2, 2)))
modelo1.add( Flatten())
modelo1.add( Dense( units = 128 , activation = 'relu'))
modelo1.add( Dense(units = 1, activation = 'sigmoid'))
# Compilamos la red
modelo1.compile(optimizer = 'adam', metrics = ['accuracy'], loss = 'binary_crossentropy')
# Entrenamos la red
size1 =(96, 96) # size1 =(64, 64)
entrenam_generador = ImageDataGenerator(rescale = 1./255, shear_range = 0.2,
                                         zoom_range = 0.25, horizontal_flip = True)
test_generador = ImageDataGenerator( rescale = 1./255 ) #
conjunto_test = test_generador.flow_from_directory( 'datos/test', #
                                                    batch_size = 32 , target_size = size1, class_mode = 'binary' )
conjunto_entrenam = entrenam_generador.flow_from_directory('datos/entrenamiento',
                                                            target_size = size1, class_mode = 'binary', batch_size = 32)
modelo1.fit_generator(conjunto_entrenam, epochs = 25, steps_per_epoch = 5000,
                      validation_data = conjunto_test, validation_steps = 1500)
# Clasificamos datos
archivo = 'gato_1.jpg'
#archivo = 'perro_1.jpg'
imagen_prueba = image.load_img(archivo, target_size = size1)
imagen_prueba = image.img_to_array(imagen_prueba)
imagen_prueba = np.expand_dims(imagen_prueba, axis = 0)
resultado = modelo1.predict(imagen_prueba)
if int(resultado[0]) == 1:
    resu_etiqueta = 'perro'
else:
    resu_etiqueta = 'gato'
print(conjunto_entrenam.class_indices)
print(resultado[0][0])
print(resu_etiqueta)

```

En la Fig. 137 se muestran las imágenes utilizadas. Se obtienen resultados correctos de clasificación.



Fig. 137 – Imágenes utilizadas para clasificación con red CNN propia.

Ejercicio 7.9

Clasificación de imágenes con red CNN y conjunto de datos CIFAR10. Lenguaje Python con librerías Tensorflow.

Se construye y se entrena una red CNN con imágenes CIFAR10. Se evalúa el entrenamiento de la red y se clasifican imágenes desde archivos.

```
import matplotlib.pyplot as plt ##
import tensorflow as tf1 ##
from tensorflow.keras import datasets, layers, models ##
##### Importamos imágenes del conjunto de datos cifar10
(imagenes_entren, etiquetas_entren) , (imagenes_test, etiquetas_test) =
datasets.cifar10.load_data()
# Normalizamos los valores de los píxeles entre 0 y 1
imagenes_entren = imagenes_entren / 255
imagenes_test = imagenes_test / 255
# Mostramos tamaño
largo_entrenamiento = len(list(imagenes_entren))
print("Cantidad de imágenes de entrenamiento: ", largo_entrenamiento)
largo_test = len(list(imagenes_test))
print("Cantidad de imágenes de testeo: ", largo_test)
# Graficamos 20 imágenes aleatorias de un total de 10 clases
import random
nombre_clases = [ 'airplane' , 'automobile' , 'bird' , 'cat' , 'deer' , 'dog' , 'frog' ,
'horse' , 'ship' , 'truck' ] # clases
plt.figure(figsize=(15,9))
for i in range(20):
    plt.subplot(4,5,i+1)
    plt.tick_params(labelbottom=False, labelleft=False)
    k = random.randint(0, largo_entrenamiento-1)
    plt.imshow(imagenes_entren[k])
    plt.xlabel(nombre_clases[etiquetas_entren[k][0]]) # índice 0 corresponde al dato
plt.show()
##### Construimos la red CNN
modelo1 = models.Sequential()
# Paso 1: conv relu, pooling
modelo1.add( layers.Conv2D( 32 , ( 3, 3 ), activation = 'relu' , input_shape = ( 32, 32, 3 ) ) )
modelo1.add(layers.MaxPooling2D(( 2, 2 )))
# Paso 2: conv relu, pooling
modelo1.add(layers.Conv2D(64, (3, 3), activation='relu'))
modelo1.add(layers.MaxPooling2D((2, 2)))
# Paso 3: conv relu
modelo1.add(layers.Conv2D(64, (3, 3), activation='relu'))
# Paso 4: conv Flatten, Dense, Dense
modelo1.add(layers.Flatten())
modelo1.add(layers.Dense(64, activation='relu'))
modelo1.add(layers.Dense(10))
# Mostramos la red
```

```

modelo1.summary()
# Compilamos la red CNN
modelo1.compile(optimizer='adam', metrics=['accuracy'],
                loss=tf1.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
iteraciones1 = 25 # Disminuir para tener entrenamiento rápido !!!
##### Entrenamos la red
historial = modelo1.fit(imagenes_entren, etiquetas_entren, epochs=iteraciones1,
                        validation_data=(imagenes_test, etiquetas_test))
# Evaluamos la precisión de la red
test_loss, test_precision = modelo1.evaluate(imagenes_test, etiquetas_test, verbose=2)
print(test_loss)
print(test_precision)
##### Clasificamos
# Clasificamos una imagen del conjunto de prueba (test)
import numpy as np # importamos librerías
from keras.preprocessing.image import img_to_array
indice1 = 45
imagen = imagenes_test[indice1]
test_image = img_to_array(imagen) #
test_image = np.expand_dims(test_image, axis = 0)
#resu_array = (modelo1.predict(test_image) > 0.5).astype("int32")
resu_array = modelo1.predict(test_image)
resu1 = np.argmax(resu_array)
print(resu1)
print("Clase predecida:", nombre_clases[resu1])
print("Clase real: ", nombre_clases[etiquetas_test[indice1][0]])
plt.figure(figsize=(4,4)); plt.imshow(imagen)
# Clasificamos una imagen desde archivo
from keras.preprocessing import image # imágenes
archivo = 'caballo1.jpg'
size1 = (32,32)
imagen_prueba1 = image.load_img(archivo, target_size = size1) #
imagen_prueba = image.img_to_array(imagen_prueba1) #
imagen_prueba = np.expand_dims(imagen_prueba, axis = 0)
imagen_prueba = imagen_prueba/255
resu_array = modelo1.predict(imagen_prueba)
resu1 = np.argmax(resu_array)
print(resu1)
print("Clase predecida:", nombre_clases[resu1])
plt.figure(figsize=(4,4)); plt.imshow(imagen_prueba1)
##### Opcional: Graficamos historial de Precisión
### Cambiamos tamaño de texto
plt.rc('font', size=15)    # default
plt.rc('axes', labelsize=15) # x , y labels
plt.rc('xtick', labelsize=14); plt.rc('ytick', labelsize=14)
plt.rc('legend', fontsize=15); plt.rc('figure', titlesize=15) # figure title
fig, ax1 = plt.subplots(figsize=(8,6))
ax1.plot(historial.history['accuracy'], label='Prec. en entrenamiento')
ax1.plot(historial.history['val_accuracy'], label = 'Prec. en validacion')
ax1.set_xlabel('Iteración')

```

```

ax1.set_ylabel('Precisión')
ax1.legend()
ax1.grid()
plt.title('Valores históricos de Precisión')

```

En la Tabla XIV se muestran los resultados de la función `modelo1.summary()`

Tabla XIV – Detalles de la red CNN

Capa	Tamaño	Parámetros
conv2d_3 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_2	(MaxPooling2 (None, 15, 15, 32)	0
conv2d_4 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_3	(MaxPooling2 (None, 6, 6, 64)	0
conv2d_5 (Conv2D)	(None, 4, 4, 64)	36928
flatten_1 (Flatten)	(None, 1024)	0
dense_2 (Dense)	(None, 64)	65600
dense_3 (Dense)	(None, 10)	650
Cantidad total de parámetros 122570		

En la Fig. 138 se muestran algunas imágenes de muestra. En la Fig. 139 se muestra la evolución del entrenamiento de la red.

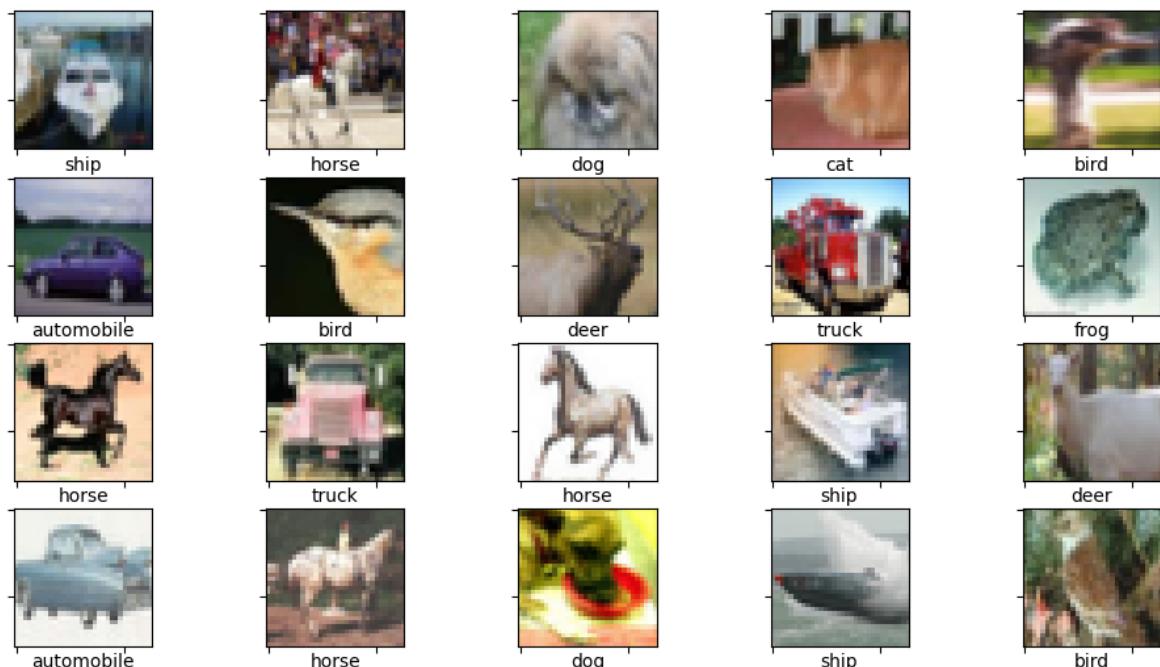


Fig. 138 – Imágenes de muestra CIFAR10

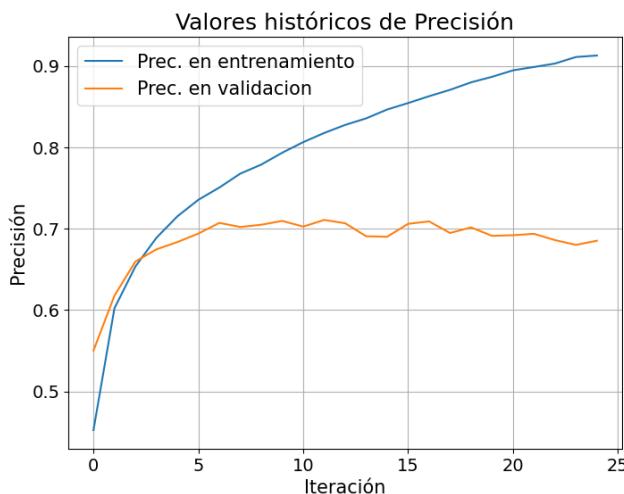


Fig. 139 – Análisis histórico del entrenamiento de datos CIFAR10

Ejercicio 7.10

Clasificación de imágenes con red CNN y conjunto de datos MNIST. Lenguaje Python con librerías Tensorflow.

Se guardar el modelo entrenado en archivo.

Se construye y se entrena una red CNN con imágenes de números MNIST. Se evalúa el progreso en el entrenamiento de la red y se clasifican imágenes. Por último, se guarda el modelo entrenado en un archivo

```
import keras #####
from keras.datasets import mnist ## base de datos
from keras.layers import Dense, Dropout, Flatten ##### importamos distintas capas
from keras.models import Sequential ## capas
from keras.layers import Conv2D, MaxPooling2D ### capas #
import matplotlib.pyplot as plt ## gráficos
import numpy as np
##### Importamos datos MNIST
(x_entren, y_entren), (x_test1, y_test1) = mnist.load_data()
x_entren = x_entren.reshape( 60000, 28, 28, 1 ) ## 28 filas y 28 columnas
x_test1 = x_test1.reshape( 10000, 28,28,1)
print('Tamaño datos de entrenamiento: ', x_entren.shape)
print('Tamaño datos de prueba: ', x_test1.shape)
# Convertimos vector de clases de salida a matrices binarias
from tensorflow.keras.utils import to_categorical ### usado para las salidas
cantidad_clases = 10 #
y_entren = to_categorical( y_entren, cantidad_clases)
y_test1 = to_categorical( y_test1, cantidad_clases)
##### Construimos la red CNN
modelo1 = Sequential()
```

```

modelo1.add( Conv2D(32, kernel_size=( 3, 3 ) , activation='relu', input_shape=( 28,28,1 )))
modelo1.add( Conv2D(64, (3, 3), activation = 'relu' ) )
modelo1.add(MaxPooling2D(pool_size=(2, 2)))
modelo1.add(Dropout(0.25))
modelo1.add(Flatten())
modelo1.add(Dense(128, activation='relu'))
modelo1.add(Dropout(0.5))
modelo1.add(Dense(cantidad_clases, activation='softmax' ) )
## Mostramos la red completa
modelo1.summary( )
## compilamos
modelo1.compile(loss=keras.losses.categorical_crossentropy,metrics=['accuracy'],
                 optimizer=keras.optimizers.Adadelta())
##### Entrenamos la red
iteraciones = 8 # 200
historial = modelo1.fit(x_entren, y_entren, batch_size= 128, epochs=iteraciones,
                        validation_data=(x_test1, y_test1))
score = modelo1.evaluate(x_test1, y_test1, verbose=0)
print('Test loss: ', score[0])
print('Precisión en el test:', score[1])
##### Clasificamos
n_muestra =48000
image1 = x_entren[n_muestra] # Elegimos una muestra
fig = plt.figure
plt.imshow(image1) ; plt.show()
nombre_clases = ['cero', 'uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis',
                  'siete', 'ocho', 'nueve']
from keras.preprocessing.image import img_to_array
test_image1 = img_to_array(image1)
test_image1 = np.expand_dims(test_image1, axis = 0)
resu_array = modelo1.predict(test_image1)
resu1 = np.argmax(resu_array)
print(resu1)
print("Clase predecida:" , nombre_clases[resu1])
print('Clase real: ', np.argmax(y_entren[n_muestra]) )
##### Opcional: Graficamos historial de Precisión
### Cambiamos tamaño de texto
plt.rcParams['font', size=15] # default
fig, ax1 = plt.subplots(figsize=(8,6))
ax1.plot(historial.history['accuracy'], label='Prec. en entrenamiento')
ax1.plot(historial.history['val_accuracy'], label = 'Prec. en validacion')
ax1.set_xlabel('Iteración') ; ax1.set_ylabel('Precisión')
ax1.legend() ; ax1.grid() ; plt.title('Valores históricos de Precisión')
##### Opcional: Guardamos el modelo completo
import tensorflow as tf
# !mkdir -p modelo_guardado
modelo1.save('modelo_guardado/mi_modelo_01')
# Cargamos el modelo y verificamos
modelo_nuevo01 = tf.keras.models.load_model('modelo_guardado/mi_modelo_01')
modelo_nuevo01.summary()

```

```
score = modelo_nuevo01.evaluate(x_test1, y_test1, verbose=0)
print('Test loss: ', score[0] ) #
print('Precisión en el test:', score[1])
```

Si se utiliza iteraciones = 200, el programa demora varias horas en entrenar el modelo. Se puede utilizar un número de iteraciones menor. Es conveniente entrenar con muchas iteraciones y guardar el modelo en un archivo (ver últimas líneas del programa). Pero también no es muy conveniente sobreentrenar los algoritmos, ya que se pierde generalización.

En la Fig. 140 se muestra una imagen de muestra. En la Fig. 141 se muestra la evolución del entrenamiento de la red.

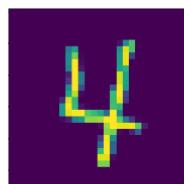


Fig. 140 – Imagen de muestra MNIST

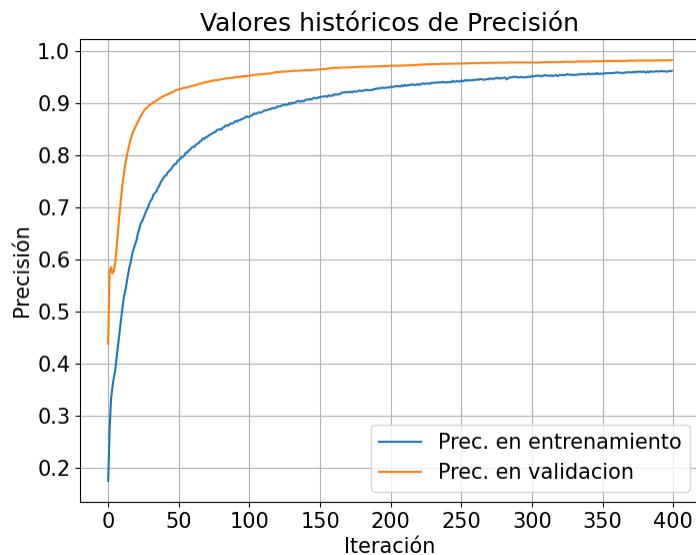


Fig. 141 – Análisis histórico del entrenamiento de datos MNIST

Ejercicio 7.11

Ejemplo simple de detección de elefantes con red CNN pre entrenada ResNet50 y VGG16. Lenguaje Python con librerías Tensorflow.

Se pide descargar una imagen de un elefante de internet y clasificarla con el siguiente programa.

```

from tensorflow.keras.applications.resnet50 import ResNet50 # Importamos red neuronal
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras.applications.resnet50 import decode_predictions
from tensorflow.keras.preprocessing import image # operación de imágenes
import numpy as np ##
import matplotlib.pyplot as plt ### pyplot
#### Cargamos una imagen de prueba, la graficamos y pre procesamos
path_archivo = 'imagen_elefante01.jpg'
img01 = image.load_img(path_archivo, target_size=(224, 224))
print(img01)
imgplot = plt.imshow(img01)
img01_proc = image.img_to_array(img01)
img01_proc = np.expand_dims(img01_proc, axis=0)
img01_proc = preprocess_input(img01_proc)
##### Cargamos Red neuronal ResNet50 y clasificamos (predicimos)
modelo1 = ResNet50( weights ='imagenet')
resultado = modelo1.predict(img01_proc)
# Decodificamos la respuesta en una lista (clase, descripción, probabilidad)
print('Valores Predecidos con ResNet50:', decode_predictions(resultado, top=5)[0] , "\n")
##### Cargamos Red neuronal VGG16 y clasificamos (predicimos)
from tensorflow.keras.applications.vgg16 import VGG16 # Importamos red VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
#modelo2 = VGG16(weights='imagenet', include_top=False)
modelo2 = VGG16(weights='imagenet', include_top=True)
resultado = modelo2.predict(img01_proc)
print('Valores Predecidos con VGG16:', decode_predictions(resultado, top=3)[0] , "\n")

```

En la Fig. 142 se muestra la imagen utilizada, se obtienen los siguientes valores de predicción (clase, descripción, probabilidad):

Valores Predecidos con ResNet50: [('n02504458', '**African_elephant**', **0.8065945**), ('n01871265', 'tusker', 0.11621026), ('n02504013', 'Indian_elephant', 0.07693434), ('n02391049', 'zebra', 8.8579545e-05), ('n01704323', 'triceratops', 2.959461e-05)]

Valores Predecidos con VGG16: [('n02504458', '**African_elephant**', **0.5155439**), ('n01871265', 'tusker', **0.46114084**), ('n02504013', 'Indian_elephant', 0.0198201)]



Fig. 142 – Imagen de prueba para clasificar con red ResNet50 y VGG16



Descarga de los códigos de los ejercicios

Referencias

- Basha, C. Z., Pravallika, B. N. L., & Shankar, E. B. (2021). An efficient face mask detector with pytorch and deep learning. *EAI Endorsed Transactions on Pervasive Health and Technology*, 7(25). <https://doi.org/10.4108/eai.8-1-2021.167843>
- Beale, M. H., Hagan, M., & Demuth, H. (2020). Deep Learning Toolbox™ User's Guide. In MathWorks. <https://la.mathworks.com/help/deeplearning/index.html>
- Beale, M. H., Hagan, M., & Demuth, H. (2020). Deep Learning Toolbox - Getting Started Guide. In MathWorks.
- Boveiri, H. R., Khayami, R., Javidan, R., & Mehdi Zadeh, A. R. (2020). Medical image registration using deep neural networks: A comprehensive review. In arXiv.
- Chaudhary, A., Chouhan, K. S., Gajrani, J., & Sharma, B. (2020). Deep Learning with PyTorch. <https://doi.org/10.4018/978-1-7998-3095-5.ch003>
- Demuth H, Beale M, Hagan M. (2018). Neural Network Toolbox™ User's Guide Neural network toolbox. MathWorks.
- Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32.
- Kaplunovich, A., & Yesha, Y. (2020). Refactoring of Neural Network Models for Hyperparameter Optimization in Serverless Cloud. *Proceedings - 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW 2020*. <https://doi.org/10.1145/3387940.3392268>
- Pytorch, Biblioteca de aprendizaje automático Pytorch, 2021. <https://pytorch.org/>
- Scikit-learn, biblioteca de software de aprendizaje automático para Python, 2021. <https://scikit-learn.org/stable/>
- Tensorflow, Biblioteca de aprendizaje automático Tensorflow, desarrollada por Google, 2021, <https://www.tensorflow.org/>
- The Mathworks, Inc. MATLAB, Version 9.6, 2019. (2019). MATLAB 2019b - MathWorks. In www.Mathworks.Com/Products/Matlab.

Capítulo VIII – Inteligencia Artificial en Dispositivos Móviles y de IoT

En este capítulo se muestran aplicaciones de inteligencia artificial para implementar en celulares, tablets, y computadoras muy pequeñas como Raspberry PI. Los últimos años surgieron aplicaciones nuevas para dispositivos móviles que tienen la ventaja de llegar a muchos usuarios y permiten adquirir imágenes y datos que no se pueden adquirir con una computadora de escritorio. Se muestran los resultados de ejemplos implementados en Java con la plataforma Android Studio y el framework Tensor Flow Lite. Este framework es bastante completo y contiene muchas aplicaciones con código de descarga gratuita. Se presentan brevemente las aplicaciones más destacadas y se muestran algunos detalles de los programas.

Se enumeran algunas aplicaciones interesantes para celulares: software para pacientes con dificultades en la visión, reconocimiento de COVID mediante el micrófono, aplicaciones de tránsito y GPS, recomendaciones personales, clasificación de imágenes y detección de objetos, detección de gestos y expresiones en imágenes, reconstrucción de imágenes, bots terapeutas, aplicación que detectan movimientos y enseñan a bailar, clasificador de audio mediante espectrogramas, etc.

Introducción

En los últimos años creció notablemente el uso de teléfonos celulares y dispositivos móviles, donde las aplicaciones de inteligencia artificial presentan un rol fundamental (Bednar, 2020), (Bokeh, 2018), (Eisenstein, 2020). Mediante la plataforma Android Studio se pueden programar aplicaciones en Java y en otros lenguajes, para descargar en celulares, tablets y minicomputadoras Raspberry PI de bajos costo y tamaño. Estas aplicaciones son multiplataformas, con compilación individual, pueden correr en Android, iOS y en Linux para Raspberry PI. Se puede programar en lenguaje Java y Kotlin, ambos interoperables entre sí. Recomendamos utilizar el framework Tensorflow Lite (Tensorflow Lite, 2021), (Artificial Intelligence with MIT App Inventor, 2021), (Abadi, 2015).

En la Fig. 143 se muestra la interfaz de Android Studio, donde se observa el diseño de la pantalla principal, para el ejemplo de aumento en resolución de imágenes (super_resolution) se accede mediante el siguiente archivo:

\examples\super_resolution\android\app\src\main\res\layout\activity_main.xml

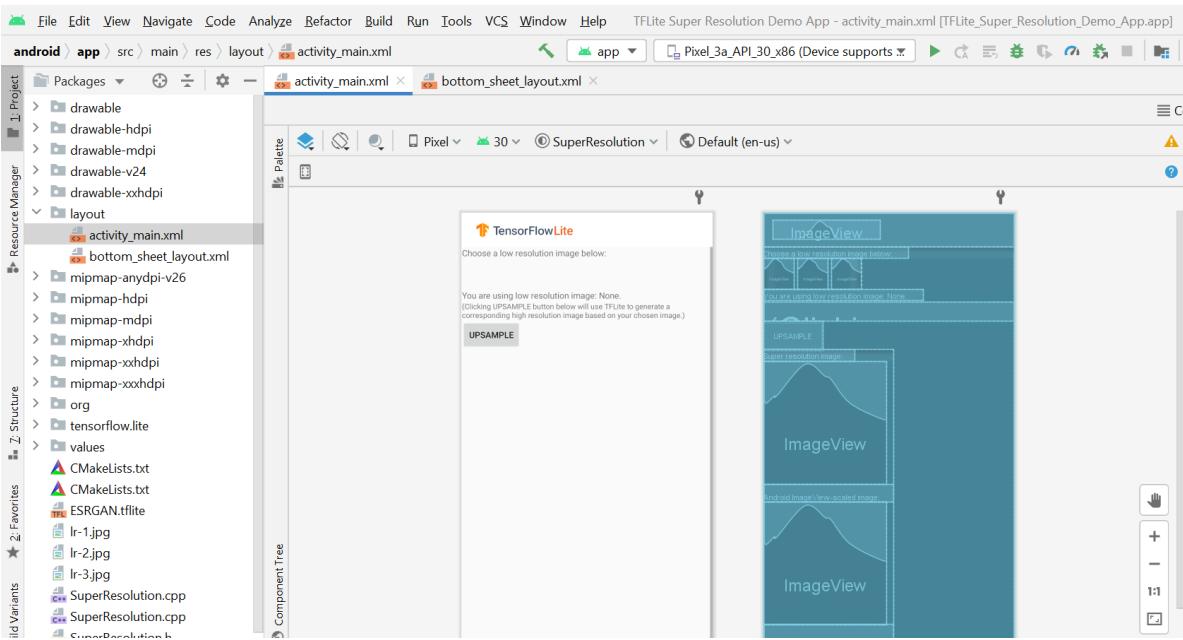


Fig. 143 – Interfaz de Android Studio y acceso al diseño de la pantalla principal

Tutoriales y ejemplos de Tensorflow para descargar

Tensorflow Lite es un entorno de trabajo (framework) de código abierto que se puede implementar en Android Studio. Permite implementar aprendizaje profundo y aprendizaje automático en dispositivos de IoT y en dispositivos móviles.

Se pueden descargar ejemplo y ayudas de las siguientes páginas web:

<https://www.tensorflow.org/lite/examples>

<https://github.com/tensorflow/examples/tree/master/lite/examples>

<http://appinventor.mit.edu/explore/ai-with-mit-app-inventor>

A continuación, mostramos algunos ejemplos de Tensorflow Lite.

- 1) Clasificador de imágenes con dígitos, ver Fig. 144.
- 2) Detector de gestos mediante webcam
- 3) Clasificación de imágenes: identifica lo que representa una imagen, se entrena para reconocer varias clases de imágenes. TensorFlow Lite proporciona modelos optimizados previamente entrenados para implementar en aplicaciones móviles, ver Fig. 145 izquierda.
- 4) Detección de objetos: dada una imagen o un video, un modelo de detección de objetos puede identificar un conjunto conocido de objetos y proporcionar información sobre sus posiciones dentro de la imagen, ver Fig. 145 derecha.

- 5) Segmentación de imágenes: mediante la segmentación se divide una imagen digital en múltiples segmentos (objetos o conjuntos de píxeles). Mediante esta técnica se simplifica y/o se cambia la representación de imágenes en algo más significativo, más fácil de analizar y de procesar.
- 6) Recomendaciones personalizadas: se utilizan ampliamente en muchos casos, como la sugerencia en recuperación de contenido multimedia, sugerencia de productos de compra, etc. En esta aplicación se respeta la privacidad del usuario.
- 7) Estimación de pose: utiliza un modelo para estimar la pose de una persona a partir de una imagen o un video mediante la estimación de las ubicaciones espaciales de las articulaciones corporales clave (puntos clave).
- 8) Respuesta inteligente: genera sugerencias de respuesta basadas en mensajes de chat. Las sugerencias están pensadas para brindar respuestas contextualmente relevantes. Por ejemplo, con un click en la pantalla ayudan al usuario a responder fácilmente a un mensaje.
- 9) Clasificación de audio: se identifica lo que representa un audio. Mediante un modelo de clasificación se reconocen varios eventos de audio. Por ejemplo: aplaudir, chasquear los dedos y escribir. TensorFlow Lite proporciona modelos optimizados previamente entrenados para aplicaciones móviles.
- 10) Identificar de comandos de voz
- 11) Transferencia de estilo artístico: crea una imagen nueva basada en 2 imágenes de entrada: una imagen representa el estilo artístico y la otra representa el contenido nuevo.
- 12) Mejora en la resolución: genera una imagen de alta resolución (HR) de su contraparte de baja resolución, esta tarea comúnmente se denomina super resolución de imagen única (SISR). Utiliza el modelo ESRGAN (ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks). Convierte una imagen que tiene baja resolución de 50 x 50 a otra imagen de alta resolución de 200 x 200 (factor de escala = 4), ver Fig. 146.
- 13) Pregunta y respuesta: utiliza un modelo de TensorFlow Lite, esta aplicación responde preguntas según el aprendizaje de una frase determinada.
- 14) Clasificación de texto: utiliza un modelo de TensorFlow Lite para clasificar un párrafo en grupos predefinidos.

Ejercicio 8.1

Clasificación de imágenes con dígitos mediante el celular

Resultados

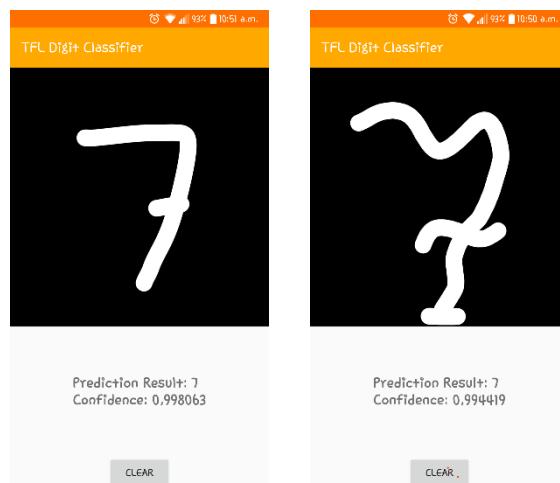


Fig. 144 – Aplicación de detección de dígitos de TensorFlow con celular

Ejercicio 8.2

Detección de objetos dentro de una imagen

Resultados

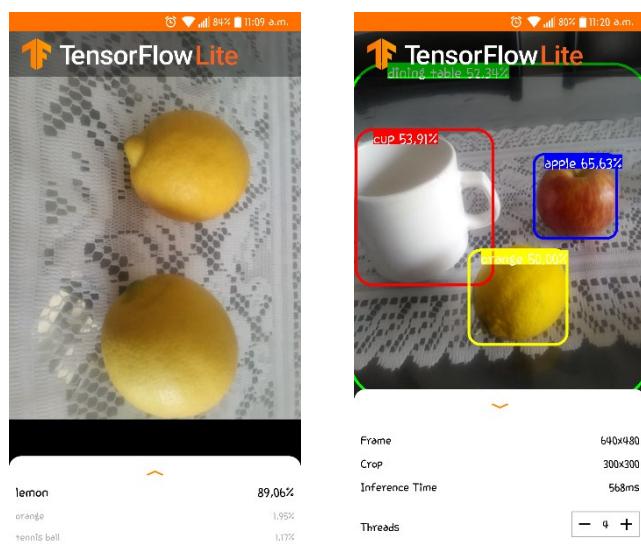


Fig. 145 – Aplicaciones de detección de imágenes con celular en Android

Ejercicio 8.3

Mejora en la resolución de imágenes

Resultados

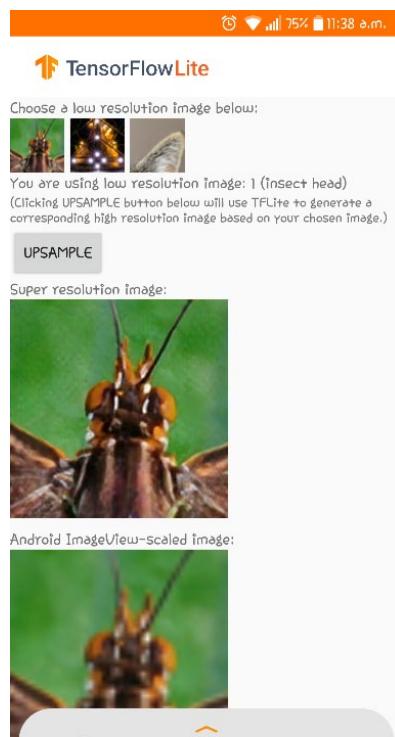


Fig. 146 – Aplicación para mejorar la resolución en imágenes

Detalles de programas de clasificación de imágenes con Tensorflow

Se pueden encontrar detalles de algoritmos de TensorFlow en las siguientes páginas web.

<https://www.tensorflow.org/tutorials/images/classification>

https://www.tensorflow.org/lite/examples/image_classification/overview

A continuación, se describe muy brevemente el programa de clasificación de imágenes.

Para ver las Clases (plantillas de programas conocidas como Class) más relevantes, ver archivo ClassifierTest, con la siguiente ruta de acceso:

\image_classification\android\app\src\androidTest\java\org\ tensorflow\lite\examples\classification\n\ClassifierTest.java

CameraActivity, ver archivo:

\image_classification\android\app\src\main\java\org\ tensorflow\lite\examples\classification\CameraActivity.java

CameraConnectionFragment, ver archivo: CameraConnectionFragment.java

ClassifierActivity, ver archivo: ClassifierActivity.java

Se importan las siguientes librerías de TensorFlow:

```
import org.tensorflow.lite.examples.classification.tflite.Classifier;
import org.tensorflow.lite.examples.classification.tflite.Classifier.Device;
import org.tensorflow.lite.examples.classification.tflite.Classifier.Model;
import org.tensorflow.lite.examples.classification.tflite.Classifier.Recognition;
```

En la clase ClassifierActivity se copian algunas líneas relevantes para crear el modelo para clasificar y para reconocer imágenes:

```
private Classifier classifier;
classifier = Classifier.create(this, model, device, numThreads);
classifier.recognizeImage(rgbFrameBitmap, sensorOrientation);
```

Diferentes modelos utilizados:

```
Model.QUANTIZED_MOBILENET
Model.QUANTIZED_EFFICIENTNET
Model.QUANTIZED_MOBILENET,
Model.FLOAT_EFFICIENTNET,
```

Modelos de redes neuronales de aprendizaje profundo en dispositivos móviles

Se nombran algunos modelos:

Mobilenet_V2_1.0_224_quant
Inception_V4_quant
DenseNet
DenseNet
MnasNet_1.3_224

En la siguiente página se pueden obtener más detalles:

https://www.tensorflow.org/lite/guide/hosted_models#image_classification

Creación de un modelo propio en TensorFlow Lite

Ejercicio 8. 4

Ejemplo para crear un modelo propio y entrenarlo con imágenes

La biblioteca “TensorFlow Lite Model Maker” simplifica el proceso de adaptación y conversión de un modelo de red neuronal de TensorFlow con datos de entrada particulares para aplicaciones de aprendizaje automático en dispositivos. En las siguientes páginas se pueden obtener más detalles:

<https://www.tensorflow.org/lite/convert>

https://www.tensorflow.org/lite/tutorials/model_maker_image_classification

Descarga de ejemplos varios en TensorFlow Lite

Ejercicio 8. 5

Ejemplos agregador de TensorFlow Lite para descarga

A continuación, se listan algunos ejemplos de programas para descargar y probar en el celular

<https://www.tensorflow.org/lite/examples>

Preguntas y respuestas (bert_qa)

Clasificación de gestos (gesture_classification)

Clasificación de imágenes (image_classification)

Segmentación de imágenes (image_segmentation)

Modelos personalizados (model_personalization)

Estimación de pose (pose_estimation y pose_net)

Recomendaciones personales (recommendation)

Respuestas a mensajes de chat (smart_reply)

Clasificación de sonidos (sound_classification)

Comandos de voz (speech_commands)

Transferencia de estilos artísticos (style_transfer)

Clasificación de texto (text_classification)

Tutoriales y ejemplos de Mit App Inventor para descargar

En la siguiente web se pueden descargar tutoriales y ejemplos de códigos de MIT (Massachusetts Institute of Technology):

<http://appinventor.mit.edu/explore/ai-with-mit-app-inventor>

A continuación, mostramos algunos ejemplos de Mit App Inventor.

- 1) Clasificación de imágenes: utiliza la cámara del celular para tomar fotos, para luego identificar y clasificar objetos. Cada clasificación viene con un nivel de confianza, expresado en porcentaje.
- 2) Clasificador de imágenes personales (gestos y expresiones): se crea y entrena un modelo propio para identificar y clasificar imágenes. Se detectan gestos y expresiones emocionales en personas (feliz, sorprendido, etc.)
- 3) Clasificador de audio personal: se entrena un modelo de audio usando grabaciones cortas de duración 1 a 2 segundos, se distinguen distintas voces. Este clasificador utiliza el espectrograma de las grabaciones de audio para crear un modelo. El Espectrograma de diferentes voces o sonidos resulta diferente, el clasificador encuentra estas diferencias para clasificar los sonidos según etiquetas preestablecidas.
- 4) Calculadora de voz: está basado en IA conversacional Alexa y Siri. Interpreta lo que uno dice y captan las intenciones. Implementa interfaz de usuario de voz (VUI) para tener una calculadora impulsada que realiza operaciones aritméticas básicas.
- 5) Bot terapeuta: aplicación que usa inteligencia artificial para desarrollo de bots de terapia usando App Inventor.
- 6) Aplicación de baile con IA: aplicación interactiva para enseñar a bailar. Cuantifica y mide los movimientos de baile para mejorar habilidades. Utiliza la nueva tecnología de IA PoseNet para rastrear puntos clave de su cuerpo, crea un modelo esquelético y desarrolla algunos métodos básicos para cuantificar, medir e identificar algunos movimientos.
- 7) Cámara con filtros faciales: Los filtros faciales de Instagram y Snapchat son muy populares en Internet. En esta aplicación, se generan filtros faciales propios. Se detectan puntos de referencia faciales. Implementa una cámara con filtro utilizando la tecnología de inteligencia artificial Facemesh.
- 8) Juego inteligente de piedra, papel y tijera: en este ejemplo se enseña a una máquina a aprender algo, cualquier cosa en realidad, y volverse, bueno, artificialmente inteligente. Utilizando el contexto de uno de los juegos para niños más simples, Piedra-Papel-Tijera, se crea un programa que permita a la máquina observar y aprender de las elecciones de juego de su usuario utilizando un Modelo de Markov para volverse lo suficientemente inteligente rápidamente para vencer repetidamente al usuario en el juego.

Algunas aplicaciones destacadas de MIT

“Go Corona Go” desarrollada por Tanisha Thaosen.

Esta aplicación se ocupa de la salud mental y cómo hacer frente a la depresión inducida por la pandemia. Tiene como objetivo ayudar a los usuarios a lidiar con la depresión inducida por la pandemia, y educarlos sobre los hechos y los mitos sobre COVID 19. Los usuarios pueden realizar un seguimiento de su estado de ánimo y expresar sus sentimientos a través de un informe diario.

“Corona Exit” desarrollada por Apostolou C., Vasileiadis I., Garnaras T., Gourmpaliotis E., Zai G., Zisopoulos D., Karadimas A., Karantakos P., Katsiamanis C., and Kostakis C.

En Grecia, durante la pandemia implementaron una restricción horaria para viajar por todo el territorio. Para que los ciudadanos se desplacen, era necesario enviar un SMS, en el caso de desplazarse por motivos laborales debían enviar el certificado de trabajo correspondiente. Esta aplicación aconseja al usuario a enviar mensajes en poco tiempo. Tiende a ayudar a personas jóvenes y mayores, independientemente de sus habilidades con los dispositivos inteligentes.



Descarga de los códigos de los ejercicios

Referencias

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., ... Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. <https://doi.org/10.5281/zenodo.4724125>
- Artificial Intelligence with MIT App Inventor, 2021, <http://appinventor.mit.edu/explore/ai-with-mit-app-inventor>
- Bednar, J. A., Crail, J., Crist-Harif, J., Rudiger, P., Brener, G., B, C., Mease, J., Signell, J., Collins, B., Stevens, J.-L., Bird, S., kbowen, maihde, Thorve, A., Ahmadia, A., Jr, B. A. B., Brandt, C. H., Tolboom, C., G., E., ... narendramukherjee. (2020). Holoviz/dashashader: Version 0.11.1 (Version v0.11.1) [Computer software]. Zenodo. <https://doi.org/10.5281/zenodo.3987379>
- Bokeh Development Team. (2018). Bokeh: Python library for interactive visualization. <https://bokeh.pydata.org/en/latest/>
- Eisenstein, M. Active machine learning helps drug hunters tackle biology. Nat Biotechnol 38, 512–514 (2020). <https://doi.org/10.1038/s41587-020-0521-4>
- Tensorflow Lite, Biblioteca de aprendizaje automático Tensorflow, desarrollada por Google, 2021, <https://www.tensorflow.org/lite/examples>

Capítulo IX – Visión Artificial

La Visión Artificial es una rama de la inteligencia artificial, puede trabajar con y sin redes neuronales. Se trata de una combinación de hardware y software para capturar y procesar imágenes. Mediante la visión artificial se desarrollan técnicas para que las máquinas puedan identificar y procesar imágenes o videos para reemplazar algunas tareas que se realizan con la visión humana. También puede realizar tareas que no realiza el ojo humano, como por ejemplo detectar detalles finos o determinadas cuantificaciones de imágenes. Se muestran aplicaciones con OpenCV y SimpleCV de detección de bordes en imágenes, detección de objetos, detección de esquinas, detección de rostros y comparación con patrones. Se trabaja con imágenes y con videos. Cabe destacar que estas aplicaciones son multiplataformas.

Introducción

La visión artificial o visión por computadora comprende diferentes técnicas para procesar y analizar imágenes reales en forma numérica con computadoras, para obtener información relevante y tomar decisiones. Existen muchas aplicaciones entre las cuales se destacan las aplicaciones médicas, automatización de procesos industriales, detección de objetos y contornos, filtrado de imágenes y sistemas de seguridad. Se pueden utilizar diferentes métodos de procesamiento de imágenes, con técnicas de extracción de características, aprendizaje automático y aprendizaje profundo. Los procesamientos se pueden realizar con y sin redes neuronales (The Mathworks, 2019).

En este capítulo solo se muestran algunas técnicas de visión artificial con ayuda de las librerías OpenCV y SimpleCV en lenguaje de programación Python. Las librerías utilizadas tienen la ventaja de ser multiplataforma, están correctamente documentadas y existen muchos ejemplos y tutoriales en internet. OpenCV se utiliza en C++, Python, JAVA, Matlab, iOS y Android (OpenCV, 2021), (Bradski, 2020), (Community, 2010).

Se muestra el enlace oficial de OpenCV:

<http://www.opencv.org/>

Utilizaremos OpenCV-Python que es la API de OpenCV desarrollada para Python (Domínguez, 2017), (OpenCV, 2015).

En la visión artificial se puede realizar distintos procesamientos de imágenes y videos, a continuación, mencionamos algunos (Nagata, 2007), (Naveenkumar, 2016)

- Lectura, escritura y visualización de imágenes y videos desde archivo o cámara de video
- Edición y conversión de imágenes (colores y espacios de colores RGB, HSV, etc.)
- Filtrado y suavizado de imágenes (por ejemplo, con las funciones cv2.filter2D(), cv2.GaussianBlur y cv2.blur de OpenCV). Filtros gradientes, Sobel Laplacianos
- Detección de bordes (algoritmo Canny)
- Búsqueda y localización de objetos dentro de una imagen con Template Matching.
- Detección de rostros con filtros tipo cascada basados en métodos Haar, desarrollados por P. Viola y M. Jones en 2001.
- Detección y extracción de características (conocido como feature detection)
- Diferentes técnicas de aprendizaje automático con métodos estadísticos y redes neuronales

Detección de bordes con OpenCV mediante función Canny

El detector Canny de OpenCV es un algoritmo popular de detección de bordes. Fue desarrollado por John F. Canny. Es un algoritmo de múltiples etapas, mostramos cada una de ellas (Phan, 2021), (Shah, 2020), (OpenCV, 2015).

- Reducción de ruido: dado que los resultados son susceptibles al ruido en la imagen, el primer paso es eliminar el ruido en la imagen con un filtro gaussiano de 5x5.
- Búsqueda de gradiente de intensidad de la imagen: la dirección del gradiente siempre resulta perpendicular a los bordes. La imagen suavizada se filtra con un kernel de Sobel tanto en dirección horizontal como vertical para obtener las derivadas de orden 1 en dirección horizontal (G_x) y en dirección vertical (G_y). A partir de estas dos imágenes, podemos encontrar el gradiente del borde y la dirección de cada píxel de la siguiente manera:

$$\text{Gradiente_bordes}(G) = \sqrt{G_x^2 + G_y^2} \quad (9.1)$$

$$\text{Ángulo} = \theta = \tan^{-1} \left(\frac{G_y}{G_x} \right) \quad (9.2)$$

- Supresión no máxima: luego de obtener la dirección y la magnitud del gradiente, se escanea la imagen completa para eliminar los píxeles sobrantes no deseados que no son parte del borde. Para ello, en cada píxel, se comprueba si el píxel es un máximo local en su vecindad en la dirección del gradiente.
- Umbral de histéresis: esta etapa se decide cuáles son los bordes verdaderos y cuáles no. Para esto, necesitamos dos valores de umbral, valor mínimo (minVal) y valor máximo (maxVal). Los bordes que tengan gradiente de intensidad superior al valor de maxVal seguramente serán bordes y los que estén por debajo de minVal seguramente no son bordes, entonces se descartan. Aquellos valores que se encuentran entre estos dos valores umbrales se pueden clasificar como bordes o no bordes dependiendo de su conectividad. Si están conectados a los píxeles de "borde seguro", se los considera parte del borde. Caso contrario, también se descartan. Esta etapa también se eliminan los ruidos de píxeles pequeños suponiendo que los bordes son líneas largas.

Mediante todos estos pasos se obtienen los bordes fuertes en la imagen.

Referencia:

https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html

Detector de objetos con SimpleBlobDetector() de OpenCV

Se detectan objetos o manchas (blobs) mediante un algoritmo simple con los siguientes pasos:

- Se convierte la imagen original en imágenes binarias aplicando diferentes umbrales desde un valor mínimo (minThreshold incluido) hasta un valor máximo (maxThreshold exclusivo) con distancia denominada umbralStep entre umbrales vecinos.

- Se extraen componentes conectados de cada imagen binaria mediante la función `findContours` y se calculan sus centros.
- Se agrupan los centros de varias imágenes binarias por sus coordenadas. Los centros cercanos forman un grupo que corresponde a un *blob*, que está controlado por el parámetro: `minDistBetweenBlobs`.
- A partir de los grupos, se estiman los centros finales de los objetos y sus radios, se devuelven las ubicaciones y tamaños de los puntos clave.

Este algoritmo puede realizar varios filtrados en los objetos. Contiene los siguientes filtros:

- Por color. Este filtro compara la intensidad de una imagen binaria en el centro de un objeto con la variable `blobColor`. Si difieren, el objeto se filtra. Se utiliza `blobColor = 0` para extraer manchas oscuras y `blobColor = 255` para extraer manchas claras.
- Por zona. Los objetos extraídos tienen un área entre un valor mínimo (llamado `minArea` inclusive) y un valor máximo (llamado `maxArea` exclusivo).
- Por circularidad. Los objetos extraídos tienen circularidad $\frac{4\pi \cdot \text{Área}}{\text{Perímetro}^2}$ entre valor mínimo (`minCircularity` inclusive) y un valor máximo (`maxCircularity` exclusivo).
- Por relación de la inercia mínima a la inercia máxima. Los objetos extraídos tienen esta relación entre un valor mínimo (`minInertiaRatio` inclusive) y un valor máximo (`maxInertiaRatio` exclusivo).
- Por convexidad. Los objetos extraídos tienen convexidad (área / área del casco convexo del blob) entre valor mínimo (`minConvexity` inclusive) y un valor máximo (`maxConvexity` exclusivo).
- Los valores predeterminados de los parámetros se ajustan para extraer manchas circulares oscuras.

Referencia:

https://docs.opencv.org/4.5.2/d0/d7a/classcv_1_1SimpleBlobDetector.html

Detección de objetos con clasificadores en cascada Haar.

Los clasificadores de objetos en cascada basados en características de Haar son métodos eficaces para detectar objetos, fueron propuestos por Paul Viola y M. Jones en su artículo " Rapid Object Detection using a Boosted Cascade of Simple Features " en 2001. Es un enfoque basado en aprendizaje automático en el que una función que trabaja en cascada se entrena con muchas imágenes positivas y negativas. Posteriormente se utiliza para detectar distintos objetos en otras imágenes.

Para la detección de rostros, el algoritmo de Haar necesita entrenarse con grandes cantidades de imágenes positivas (imágenes de rostros) y con imágenes negativas (imágenes sin rostros). Entonces se necesitan extraer características. Para ello, se utilizan las funciones de Haar como núcleo convolucional. Cada una de las características son valores únicos obtenidos de restar la suma de todos los píxeles debajo del rectángulo blanco y la suma de los píxeles debajo del rectángulo negro.

Luego, los tamaños y las ubicaciones posibles de cada kernel se utilizan para calcular muchas características (por ejemplo, una ventana de 24x24 da como resultado de 160000 funciones). Para el cálculo de cada característica, se necesita encontrar la suma de todos los píxeles debajo de los rectángulos blancos y negros. Se utiliza el cálculo de la imagen integral. No importa cuán grande sea

la imagen, se realizan los cálculos para un píxel dado a una operación que involucra solo cuatro píxeles.

Pero entre todas estas características que se calculan, la mayoría son irrelevantes. Se seleccionan las más importantes mediante un método llamado Adaboost. Con este método se seleccionan las características con menor tasa de error, lo que significa que se toman solo las características que clasifican con mayor precisión las imágenes de rostro faciales y las no faciales. Entonces, el clasificador final es la suma ponderada de los clasificadores débiles. Se denominan débiles porque por sí solo no pueden clasificar las imágenes, pero junto con otros forman un clasificador fuerte. Incluso solo 200 funciones proporcionan detección con una precisión del 95%. Se puede realizar una configuración final con alrededor de 6000 funciones. Se reduce de más de 160000 funciones a solo 6000 funciones.

Entonces se toma una imagen, y se toman ventanas de 24x24 y se aplican 6000 funciones. Para no tener tanta carga computacional, se concentra solo en las regiones donde puede haber rostros. En una imagen, muchas veces la mayor parte de la imagen es una región sin rostro. Para ello se introduce el concepto nuevo de Cascada de Clasificadores. En vez de aplicar las 6000 características en la ventana, las características se pueden agrupar en varias etapas de clasificadores que se aplican por separado una por una. Generalmente, las primeras etapas contendrán muchas menos funciones. Si una ventana falla en la primera etapa, se descarta. Si pasa, se aplica la segunda etapa de características y así sucesivamente. Las ventanas que pasan por todas las etapas son regiones de la cara.

Se explicó en forma simple e intuitiva cómo funciona la detección de rostros Viola-Jones. Se recomienda leer la referencia para obtener más detalles.

Referencia:

https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html

Detección de esquinas con detector de Harris (Harris corner Detection)

En los algoritmos de visión artificial, las esquinas corresponden a regiones de la imagen con una gran variación de intensidad en todas sus direcciones. Chris Harris y Mike Stephens hicieron un primer intento para encontrar estas esquinas en su artículo: A Combined Corner and Edge Detector, en 1988. Debido al autor el algoritmo se llama Harris Corner Detector. Esta idea se expresa mediante una ecuación. Básicamente, indica la diferencia de la intensidad para cierto desplazamiento de (u, v) en todas las direcciones. Esto se puede expresar mediante la siguiente ecuación:

$$E(u, v) = \sum_{x,y} w(x, y) \cdot [I(x + u, y + v) - I(x, y)]^2 \quad (9.3)$$

$w(x, y)$: función de ventana

$I(x + u, y + v)$: intensidad desplazada

$I(x, y)$: intensidad

La función de ventana es una ventana rectangular o una ventana gaussiana que asigna pesos a los píxeles que se encuentran debajo. Las esquinas corresponden a los valores máximos de $E(u, v)$. Se buscan los valores máximos del segundo término. Mediante la expansión de Taylor y algunos pasos matemáticos, obtenemos la ecuación final:

$$E(u, v) \cong [u \ v]. M. \begin{bmatrix} u \\ v \end{bmatrix} \quad (9.4)$$

$$M = \sum_{x,y} w(x, y). \begin{bmatrix} I_x \cdot I_x & I_x \cdot I_y \\ I_x \cdot I_y & I_y \cdot I_y \end{bmatrix} \quad (9.5)$$

Donde I_x e I_y son las derivadas respecto de x y de y , se pueden calcular mediante la función Sobel de OpenCV. Luego se plantea R con la siguiente ecuación:

$$R = \det(M) - k. (\text{traza}(M))^2 \quad (9.6)$$

Donde $\text{traza}(M) = \lambda_1 + \lambda_2$, $\det(M) = \lambda_1 \cdot \lambda_2$

λ_1 y λ_2 : autovalores de la matriz M

Las magnitudes de estos autovalores deciden si una región es una esquina, un borde o un plano.

$$R = \lambda_1 \cdot \lambda_2 - k. (\lambda_1 + \lambda_2)^2 \quad (9.7)$$

Cuando $R < 0$, esto sucede cuando $\lambda_1 \gg \lambda_2$ o viceversa, la región corresponde a un borde.

Cuando R es grande, lo que sucede cuando λ_1 y λ_2 son grandes y de valores similares entre sí, la región es una esquina.

Cuando $|R|$ es pequeño, esto se cumple cuando λ_1 y λ_2 son bajos, entonces la región es plana.

Por tanto, el resultado del detector de bordes de Harris es una imagen en escala de grises que contiene estas puntuaciones. Con este método se utilizan umbrales para analizar las esquinas de la imagen.

Referencia:

https://docs.opencv.org/4.5.0/dc/d0d/tutorial_py_features_harris.html

Shi-Tomasi con función goodFeaturesToTrack

En 1994, J. Shi y C. Tomasi implementaron una modificación leve al detector de Harris visto anteriormente. Publicaron un artículo *Good Features to Track*, que muestra mejoras al detector de Harris. El algoritmo de Shi-Tomasi en OpenCV se implementa con la función `goodFeaturesToTrack`, donde se calcula:

$$R = \min(\lambda_1, \lambda_2) \quad (9.8)$$

Siendo λ_1 y λ_2 los autovalores de la matriz M vistos en el detector de Harris.

Si R es mayor que un valor umbral, se considera una esquina. Es decir, solo cuando λ_1 y λ_2 están por encima de un valor mínimo, λ_{min} , se considera una esquina.

Referencia:

https://docs.opencv.org/4.5.2/d4/d8c/tutorial_py_shi_tomasi.html

Comparación con patrones (Template matching)

Este algoritmo busca la posición de una imagen patrón (template), dentro de otra imagen de mayor tamaño. Busca partes de una imagen que sean similares a un patrón de búsqueda. En OpenCV se utiliza la función: cv2.matchTemplate. El algoritmo desplaza la imagen patrón sobre la imagen original, similar a una convolución 2D, para buscar el patrón. Como resultado se obtiene otra imagen en escala de grises. Cada píxel de esta imagen indica cuánto se parece a la imagen patrón. Luego se aplica la función cv2.minMaxLoc() para poder localizar el patrón buscado en la imagen.

El algoritmo compara una imagen patrón o plantilla con regiones de otra imagen superponiendo ambas imágenes. La función se desliza a través de la imagen, compara los datos superpuestos de tamaño $w \times h$ con el patrón utilizando el método especificado y almacena los resultados de la comparación como resultado.

En la Tabla XV se muestran distintas técnicas utilizadas en OpenCV. Donde I denota imagen, T patrón o plantilla, R resultado. La suma se realiza sobre la plantilla y / o la imagen:

$$x' = 0, \dots, w - 1 ; y' = 0, \dots, h - 1$$

Una vez que la función finaliza la comparación, las mejores coincidencias se pueden encontrar como mínimos globales (cuando se usó TM_SQDIFF) o máximos (cuando se usó TM_CCORR o TM_CCOEFF) usando la función minMaxLoc. En el caso de una imagen en color, la suma de la plantilla en el numerador y cada suma en el denominador se realiza en todos los canales y se utilizan valores medios separados para cada canal de color. Es decir, la función puede tomar una plantilla de color y una imagen en color. El resultado del algoritmo es una imagen de un solo canal, que es más fácil de analizar

Tabla XV – Técnicas de comparación de métodos *matchTemplate* utilizadas en OpenCV

Nombre	Fórmulas utilizadas
TM_SQDIFF	$R(x, y) = \sum_{x',y'} [T(x', y') - I(x + x', y + y')]^2$ (9.9)
TM_SQDIFF_NORMED	$R(x, y) = \frac{\sum_{x',y'} [T(x', y') - I(x + x', y + y')]^2}{\sqrt{\sum_{x',y'} T(x', y')^2 \cdot \sum_{x',y'} I(x + x', y + y')^2}}$ (9.10)
TM_CCORR	$R(x, y) = \sum_{x',y'} [T(x', y') \cdot I(x + x', y + y')]$ (9.11)
TM_CCORR_NORMED	$R(x, y) = \frac{\sum_{x',y'} [T(x', y') \cdot I(x + x', y + y')]}{\sqrt{\sum_{x',y'} T(x', y')^2 \cdot \sum_{x',y'} I(x + x', y + y')^2}}$ (9.12)
TM_CCOEFF	$R(x, y) = \sum_{x',y'} [T'(x', y') \cdot I'(x + x', y + y')]$ (9.13) $T'(x', y') = T(x', y') - \frac{1}{w.h} \cdot \sum_{x'',y''} T(x'', y'')$ (9.14) $I'(x + x', y + y') = I(x + x', y + y') - \frac{1}{w.h} \cdot \sum_{x'',y''} I(x + x'', y + y'')$ (9.15)
TM_CCOEFF_NORMED	$R(x, y) = \frac{\sum_{x',y'} [T'(x', y') \cdot I'(x + x', y + y')]}{\sqrt{\sum_{x',y'} T'(x', y')^2 \cdot \sum_{x',y'} I'(x + x', y + y')^2}}$ (9.16)

Referencia:

https://docs.opencv.org/3.1.0/df/dfb/group__imgproc__object.html#ga3a7850640f1fe1f58fe91a2d7583695d

Ejercicios

Ejercicio 9.1

Ejemplo para contar manzanas con función detector de bordes Canny.

En este programa se lee una imagen, se convierte a escala de grises, se realiza un suavizado Gaussiano, se detectan los bordes, se buscan los contornos y se muestran los resultados, ver Fig. 147.

```
# Ejemplo para contar manzanas con función Canny. Se pueden contar distintos objetos
# Se utiliza esta imagen para contar manzanas: manzanas2.jpg
import cv2
# Cargamos y mostramos la imagen original
img_orig = cv2.imread("imag1/manzanas2.jpg")
cv2.imshow("Imagen original", img_orig);cv2.waitKey(1)
```

```

# Convertimos a escala de grises
img_gris = cv2.cvtColor(img_orig, cv2.COLOR_BGR2GRAY)
cv2.imshow("Imagen grises", img_gris) ;cv2.waitKey(1)
# Aplicamos suavizado Gaussiano
img_suavizada = cv2.GaussianBlur(img_gris, (5,5), 0)
cv2.imshow("Imagen suavizada", img_suavizada) ;cv2.waitKey(1)
# Detectamos los bordes con Canny
img_canny = cv2.Canny(img_suavizada, 5, 290)
cv2.imshow("Bordes", img_canny) ;cv2.waitKey(1)
img_canny_inv = cv2.bitwise_not(img_canny) # Invertimos
cv2.imshow("Bordes Invertidos", img_canny_inv) ;cv2.waitKey(1)
# Buscamos los contornos
(contornos1,_) = cv2.findContours(img_canny.copy(), cv2.RETR_EXTERNAL,
                                  cv2.CHAIN_APPROX_SIMPLE)
# Mostramos cantidad de manzanas
texto1 = "Se encontraron {} manzanas".format(len(contornos1))
print(texto1)
# Agregamos contornos a la imagen original
cv2.drawContours(img_orig, contornos1,-1,(255,0,0), 6)
# Agregamos texto en la imagen
posic = (5,35) # posición
cv2.putText(img_orig, texto1, posic, thickness = 2,fontFace =
            cv2.FONT_HERSHEY_COMPLEX, fontScale = 0.8, color = (255,0,0))
# Mostramos y guardamos la imagen con los resultados
cv2.imshow("Resultados y Contornos", img_orig) ; cv2.waitKey(0)
cv2.imwrite('Resultado.jpg', img_orig)
cv2.destroyAllWindows()

```

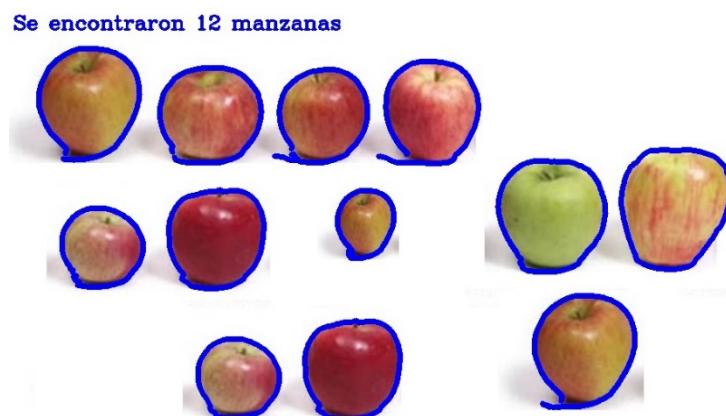


Fig. 147 – Identificación de manzanas

Ejercicio 9.2

Ejemplo para contar monedas mediante función SimpleBlobDetector() de OpenCV

A continuación, se muestra el programa, en la Fig. 148 se muestran los resultados.

```
# Ejemplo para contar monedas. Se pueden contar distintos objetos
import cv2
# Cargamos y mostramos la imagen original
import numpy as np
def inicio_detector1_blob():
    parametros = cv2.SimpleBlobDetector_Params()
    parametros.minThreshold = 10
    parametros.maxThreshold = 255
    parametros.filterByArea = True
    parametros.minArea = 400
    parametros.filterByCircularity = False
    parametros.filterByConvexity = False ; parametros.minConvexity = 0.85
    parametros.filterByInertia = False ; parametros.minInertiaRatio = 0.01
    parametros.filterByColor = True
    #detector = cv2.SimpleBlobDetector(parametros)
    detector = cv2.SimpleBlobDetector_create(parametros)
    return detector
detector = inicio_detector1_blob()
# Se puede cargar el detetector con los parámetros por defecto.
#detector = cv2.SimpleBlobDetector()
img_orig1 = cv2.imread("imag1/monedas3.jpg", cv2.IMREAD_GRAYSCALE)
cv2.imshow("Original", img_orig1)
# Detectamos blobs.
keypoints = detector.detect(img_orig1)
# Graficamos blobs con círculos
# cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS verificamos si el tamaño de círculos
corresponde al tamaño blobs
img_y_keypoints = cv2.drawKeypoints(img_orig1, keypoints, np.array([]), (255,0,0),
                                    cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
# Mostramos keypoints
cv2.imshow("Keypoints", img_y_keypoints)
cv2.imwrite('figura02a_1.jpg', img_y_keypoints)
# Graficamos solo keypoints
img_keypoints1 = cv2.drawKeypoints(img_orig1-img_orig1, keypoints, np.array([]), (255,255,255),
                                    cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
img_keypoints1 = cv2.bitwise_not(img_keypoints1) # Invertimos
cv2.imshow("Keypoints 2", img_keypoints1) ;
cv2.imwrite('figura02a_2.jpg', img_keypoints1)
cv2.waitKey(0) ; cv2.destroyAllWindows()
```

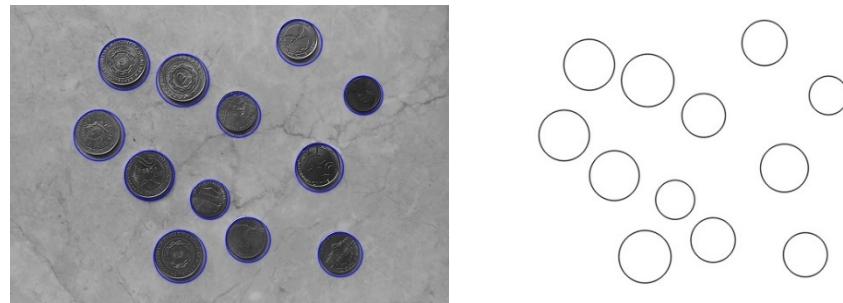


Fig. 148 – Identificación de monedas con función SimpleBlobDetector . Izq.) Imagen completa, der.) contornos

Ejercicio 9.3

Ejemplo para contar monedas con la función de detección de bordes Canny.

En la Fig. 149 se muestran los resultados del programa.

```
import cv2
# Cargamos la imagen
img_original = cv2.imread("imag2/monedas2.jpg")
cv2.imshow("Imagen original", img_original) #
# Convertimos imagen a escala de grises #
img_gris = cv2.cvtColor( img_original, cv2.COLOR_BGR2GRAY)
cv2.imshow("Imagen grises", img_gris)
# Aplicamos suavizado Gaussiano
img_suavizada = cv2.GaussianBlur(img_gris, (5,5), 0)
cv2.imshow("Imagen suavizada", img_suavizada)
# Detectamos los bordes con Canny
img_canny = cv2.Canny(img_suavizada, 5, 360) # 5,290 / 50,150
cv2.imshow("Bordes", img_canny)
img_canny_inv = cv2.bitwise_not(img_canny) # Invertimos
cv2.imshow("Bordes Invertidos", img_canny_inv)
# Buscamos los contornos
(contornos1,_) = cv2.findContours(img_canny.copy(), cv2.RETR_EXTERNAL,
                                 cv2.CHAIN_APPROX_SIMPLE)
# Mostramos cantidad de manzanas
texto1 = "Se encontraron {} monedas".format(len(contornos1))
print(texto1)
# Agregamos contornos a la imagen original
cv2.drawContours(img_original, contornos1,-1,(255,0,0), 3)
# Agregamos texto en la imagen
posic = (5,20) # posición
cv2.putText(img_original, texto1, posic, thickness = 2,
            fontFace = cv2.FONT_HERSHEY_COMPLEX, fontScale = 0.7, color = (255,0,0))
# Mostramos y guardamos la imagen con los resultados
cv2.imshow("Resultados y Contornos", img_original)
cv2.imwrite('figura03.jpg', img_original)
cv2.waitKey(0) ; cv2.destroyAllWindows()
```

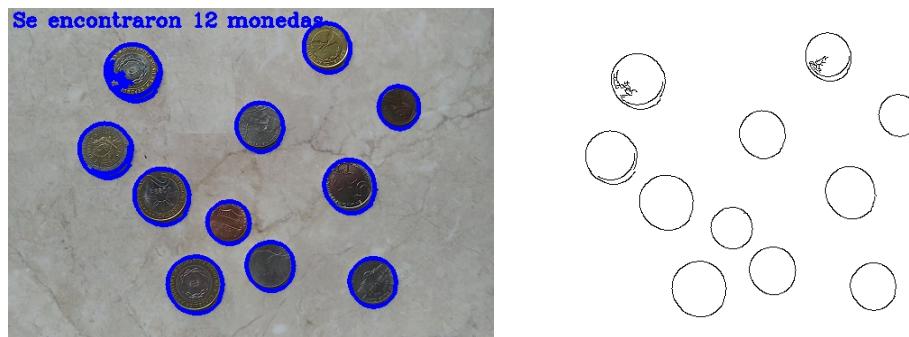


Fig. 149 – Identificación de monedas con función Canny . Izq.) Imagen completa, der.) contornos

Ejercicio 9.4

Ejemplo de detección de rostros en una imagen con *Haar Cascade Classifier*.

La detección de rostros no es una tarea simple para la computadora. Se utiliza un clasificador entrenando, llamado *Haar Cascade Classifier*. Se utilizan técnicas aprendizaje automático, donde el clasificador fue entrenado con una gran cantidad de imágenes para que pueda distinguir entre presencia o no presencia de rostros. Primero se realiza una extracción de características de las imágenes, para luego detectar rostros. Se utilizan librerías de OpenCV, se muestran los resultados en la Fig. 150.

```
import cv2
# Cargamos clasificador de rostros pre entrenado
clasific_rostros = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
# Leemos imagen desde archivo
img01 = cv2.imread('rugby1.jpg')
# Convertimos a escala de grises
img01_gris = cv2.cvtColor(img01, cv2.COLOR_BGR2GRAY)
# Detectamos rostros
rostros = clasific_rostros.detectMultiScale(img01_gris, 1.2, 4)
# Graficamos los rectángulos en los rostros
for (x1, y1, w1, h1) in rostros:
    cv2.rectangle(img01, (x1, y1), (x1 + w1, y1 + h1), (255, 255, 255), 3)
# Mostramos imagen original con rectángulos
cv2.imshow('img', img01)
cv2.imwrite('figura04.jpg', img01)
cv2.waitKey(0) ; cv2.destroyAllWindows()
```



Fig. 150 – Detección de rostros

Ejercicio 9.5

Ejemplo de detección de rostros en un archivo de video con *Haar Cascade Classifier*.

Se utiliza el mismo clasificador del ejercicio anterior, pero aplicado a un video. Se puede usar un archivo de video o la webcam de la computadora.

```
import cv2
# Cargamos clasificador de rostros pre entrenado
clasific_rostros = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
# Capturamos video desde webcam.
video1 = cv2.VideoCapture(0)
# video1 = cv2.VideoCapture('archivo_video.mp4') # Se puede leer archivo de video!
while True:
    _, img01 = video1.read() # Leemos el video
    # Convertimos a grises
    img01_grises = cv2.cvtColor(img01, cv2.COLOR_BGR2GRAY)
    # Detectamos rostros
    rostros = clasific_rostros.detectMultiScale(img01_grises, 1.1, 4)
    # Graficamos los rectángulos en los rostros
    for (x1, y1, w1, h1) in rostros:
        cv2.rectangle(img01, (x1, y1), (x1+w1, y1+h1), (255, 220, 0), 2)
    # Mostramos imagen
    cv2.imshow('img', img01)
    ## Termina el programa si se presiona ESC
    tecla = cv2.waitKey( 30 ) & 0xff
    if tecla == 27:
        break
    video1.release()
    cv2.waitKey(0) ; cv2.destroyAllWindows()
```

Ejercicio 9.6

Ejemplo de detección de sonrisas en una imagen con *Haar Cascade Classifier*.

Se detectan sonrisas mediante el detector de sonrisas de *Haar Cascade*. En la Fig. 151 se muestran los resultados del clasificador.

```
import os
import cv2
# Cargamos imagen
img01 = cv2.imread("sonrisas2.jpg")
# Cargamos el clasificador para detectar sonrisas
cv2_path = os.path.dirname(os.path.abspath(cv2.__file__))
archivo_modelo_smile = cv2_path + os.sep + '\data\haarcascade_smile.xml'
clasif_smile = cv2.CascadeClassifier(archivo_modelo_smile)
# Podemos cargar el modelo directamente si se encuentra en la misma carpeta
#clasif_smile = cv2.CascadeClassifier('haarcascade_smile.xml')
# Detectamos sonrisas
escala = 1.75 #1.8
# Parámetro que indica cuántos vecinos se debe analizar en cada rectángulo
cant_vecinos = 8 # 13
#sonrisas = clasif_smile.detectMultiScale(img01, scaleFactor = escala, minNeighbors=13)
sonrisas = clasif_smile.detectMultiScale(img01, scaleFactor = escala,
                                         minNeighbors = cant_vecinos)
for (x1, y1, w1, h1) in sonrisas:
    cv2.rectangle(img01, (x1, y1), ((x1 + w1), (y1 + h1)), (0, 0, 0), 5)
cv2.imshow("Sonrisas detectadas", img01)
cv2.imwrite('Resultado.jpg', img01)
cv2.waitKey(0) ; cv2.destroyAllWindows()
```



Fig. 151 – Detección de sonrisas

Ejercicio 9.7

Detección de esquinas mediante la función corner Harris de OpenCV

En la Fig. 152 se muestra un ejemplo de detección de esquinas en un tablero de ajedrez.

```
import numpy as np # Librería numpy
import cv2 # Librería OpenCV
titulo_fuente = 'Título Imagen Original' # fuente
titulo_bordes = ' Título Bordes detectados'
umbral_maximo = 255 #
def cornerHarris_actualizar(valor):
    umbral = valor # valor nuevo de umbral
    # Detección de esquinas y parámetros del algoritmo
    tamaño_apertura = 3 ; tamaño_bloque = 2 ; k = 0.038
    dst1 = cv2.cornerHarris(img01_gris, tamaño_bloque, tamaño_apertura, k)
    # Normalizamos
    dst1_norm = np.empty(dst1.shape, dtype=np.float32)
    cv2.normalize(dst1, dst1_norm, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX)
    dst1_norm_scaled = cv2.convertScaleAbs(dst1_norm)
    # Graficamos círculos alrededor de los bordes
    for ii in range(dst1_norm.shape[0]):
        for jj in range(dst1_norm.shape[1]):
            if int(dst1_norm[ii,jj]) > umbral:
                cv2.circle(dst1_norm_scaled, (jj,ii), 4, (0), 1)
    # Mostramos los resultados
    cv2.namedWindow(titulo_bordes)
    cv2.imshow(titulo_bordes, dst1_norm_scaled)
    img01_orig = cv2.imread("tablero1.jpg")
    img01_gris = cv2.cvtColor(img01_orig, cv2.COLOR_BGR2GRAY)
    # Creamos figura con Trackbar (barra de desplazamiento)
    cv2.namedWindow(titulo_fuente)
    umbral = 80 # valor inicial
    cv2.createTrackbar('umbral: ', titulo_fuente, umbral, umbral_maximo, cornerHarris_actualizar)
    # Graficamos imagen original en figura anterior
    cv2.imshow(titulo_fuente, img01_orig)
    cornerHarris_actualizar(umbral)
    cv2.waitKey() ;
    # cv2.imwrite('Resultado.jpg', dst1_norm_scaled)
    cv2.destroyAllWindows()
```

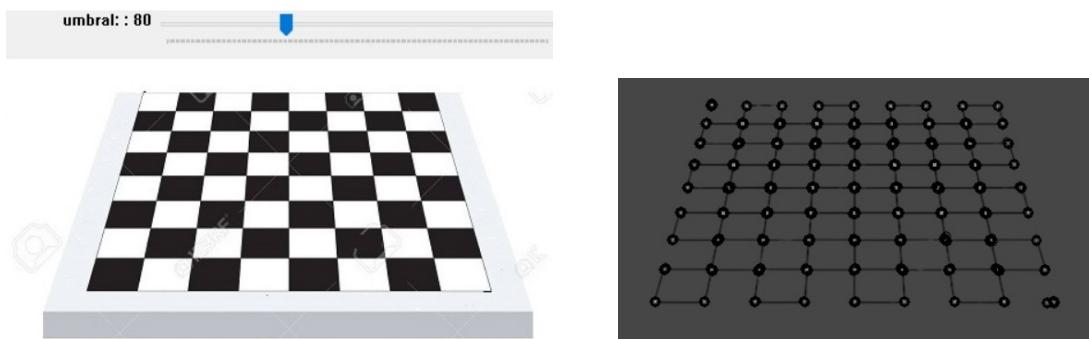


Fig. 152 – Detección de esquinas. Izq.) Tablero original, der.) Esquinas reconocidas

Ejercicio 9.8

Detección de esquinas usando el método Shi-Tomasi con función goodFeaturesToTrack

En la Fig. 153 y Fig. 154 se muestran los resultados de la detección.

```
import cv2 # Importamos librería OpenCV
import numpy as np
import random as rng1
rng1.seed(1000)
titulo1 = 'Imagen Original'
def goodFeaturesToTrack_Actualizar(valor):
    max_esquinas1 = max(valor, 1)
    # Parámetros del detector
    distancia_min = 8 ; calidad = 0.015 ;
    nn=3 # tamaño_bloque y gradiente
    kk = 0.04 # Parámetro del algoritmo
    # Copiamos imagen original y detectamos esquinas
    img_copia = np.copy(img_orig)
    esquinas = cv2.goodFeaturesToTrack(img_gris, max_esquinas1, calidad, distancia_min, None, \
        blockSize=nn, gradientSize=nn, useHarrisDetector=False, k=kk )
    # Graficamos esquinas detectadas y mostramos resultados
    print('Esquinas detectadas: ', esquinas.shape[0])
    radio = 5 #4
    for i in range(esquinas.shape[0]):
        cv2.circle(img_copia, (int(esquinas[i,0,0]), int(esquinas[i,0,1])), radio,
            (rng1.randint(0,256), rng1.randint(0,256), rng1.randint(0,256)), cv2.FILLED)
    cv2.namedWindow(titulo1)
    cv2.imshow(titulo1, img_copia)
# Cargamos imagen original y convertimos a escala de grises
#archivo = 'imagen autopista1.jpg'
archivo = 'figuras geom1.jpg'
img_orig = cv2.imread( archivo )
img_gris = cv2.cvtColor(img_orig, cv2.COLOR_BGR2GRAY)
# Creamos figura con barra
```

```

cv2.namedWindow(titulo1)
ini_max_esquinas = 77 # valor inicial
maxBarra = 200
cv2.createTrackbar('Umbral: ', titulo1, ini_max_esquinas, maxBarra,
goodFeaturesToTrack_Actualizar)
cv2.imshow(titulo1, img_orig)
goodFeaturesToTrack_Actualizar(ini_max_esquinas)
cv2.waitKey() ; cv2.destroyAllWindows()

```

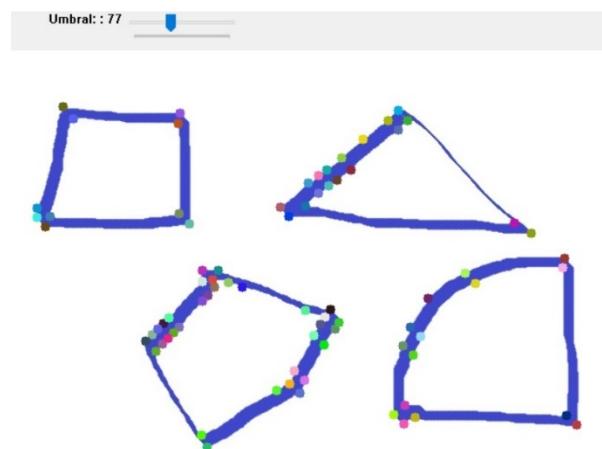


Fig. 153 – Detección de esquinas en figuras geométricas con método Shi-Tomasi



Fig. 154 – Detección de esquinas y autos con método Shi-Tomasi

Ejercicio 9.9

Detección de bordes con webcam y función Canny.

En la Fig. 155 se muestra un ejemplo de detección de bordes utilizando video de una webcam.

```

import cv2
webcam_id1 = 0
camara1 = cv2.VideoCapture(webcam_id1)
# Umbrales y apertura del detector Canny

```

```

umbral_1 = 50
umbral_2 = 150
apertura = 3
# Leemos imágenes desde webcam y detectamos bordes
while(camara1.isOpened()):
    ret, frame = camara1.read() # captura 1 imagen
    if ret == True:
        # Detectamos bordes
        bordes1 = cv2.Canny(frame, umbral_1, umbral_2, apertura)
        # Mostramos imagen original, imagen con bordes y bordes invertidos
        cv2.imshow("Original", frame)
        cv2.imshow("Detección de bordes", bordes1)
        bordes1_inv = cv2.bitwise_not(bordes1) # Invertimos
        cv2.imshow("Detección de bordes Inv", bordes1_inv)
        # Presionar tecla a para salir
        if cv2.waitKey( 1 ) & 0xFF == ord('a'):
            break
    else:
        break
# Finalizamos
camara1.release()
cv2.destroyAllWindows()

```



Fig. 155 – Detección de bordes

Ejercicio 9.10

Búsqueda y detección de objetos o imágenes dentro de otra imagen.

En la Fig. 156 se muestran los resultados donde se observa la imagen buscada en recuadro.

```

import cv2 ## librería OpenCV ##
from matplotlib import pyplot as plt ##
img_completa = 'rugby1.jpg' # imagen 1 completa
img1 = cv2.imread(img_completa, 0)
img2 = img1.copy()
archivo_patron = 'rugby1_templ.jpg' # imagen patrón a buscar

```

```

img_patron = cv2.imread(archivo_patron, 0)
w, h = img_patron.shape[::-1]
# Se utilizan 6 métodos y se comparan
lista_metodos = [ 'cv2.TM_CCOEFF', 'cv2.TM_CCOEFF_NORMED' , 'cv2.TM_SQDIFF',
                  'cv2.TM_SQDIFF_NORMED' , 'cv2.TM_CCORR', 'cv2.TM_CCORR_NORMED']
for met in lista_metodos:
    img2_copy = img2.copy()
    metodo = eval(met)
    # Buscamos coincidencias con el patrón
    resultado1 = cv2.matchTemplate( img2_copy , img_patron, metodo )
    min_val1, max_val1, min_loc1, max_loc1 = cv2.minMaxLoc(resultado1)
    # Para métodos TM_SQDIFF or TM_SQDIFF_NORMED, usamos mínimo
    if metodo in [cv2.TM_SQDIFF, cv2.TM_SQDIFF_NORMED]:
        top_left1 = min_loc1
    else:
        top_left1 = max_loc1
    bottom_right1 = (top_left1[0] + w, top_left1[1] + h)
    # Mostramos resultados
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
    fig.suptitle(met)
    ax1.imshow(resultado1,cmap = 'gray')
    ax1.set_title('Resultado coincidente'), ax1.set_xticks([]), ax1.set_yticks([])
    # Agregamos rectángulo a la imagen
    cv2.rectangle(img2_copy, top_left1, bottom_right1, 255, 4) #
    ax2.imshow( img2_copy, cmap = 'gray' ) ## escala de grises ##
    ax2.set_title('Detección'), ax2.set_xticks([]), ax2.set_yticks([])
    plt.show()

```



Fig. 156 – Detección de una imagen dentro de otra imagen. Izq.) Imagen procesada, der.) resultado obtenido con imagen remarcada.



Descarga de los códigos de los ejercicios

Referencias

- Bradski, G. (2000). The OpenCV Library. Dr. Dobb's Journal of Software Tools.
- Community, O. (2010). The OpenCV Reference Manual. October.
- Domínguez, C., Heras, J., & Pascual, V. (2017). IJ-OpenCV: Combining ImageJ and OpenCV for processing images in biomedicine. Computers in Biology and Medicine, 84. <https://doi.org/10.1016/j.combiomed.2017.03.027>
- Nagata, M. (2007). OpenCV. Kyokai Joho Imeji Zasshi/Journal of the Institute of Image Information and Television Engineers, 61(11). <https://doi.org/10.3169/itej.61.1602>
- Naveenkumar, M., & Ayyasamy, V. (2016). OpenCV for Computer Vision Applications. Proceedings of National Conference on Big Data and Cloud Computing (NCBDC'15), March 2015.
- OpenCV. (2015). OpenCV Tutorials. OpenCV 2.4.11.0 Documentation: Available at Https://Docs.Opencv.Org/2.4/Opencv_tutorials.Pdf.
- OpenCV, 2021, <https://docs.opencv.org>
- Phan, T. H., Tran, D. C., & Hassan, M. F. (2021). Vietnamese character recognition based on cnn model with reduced character classes. Bulletin of Electrical Engineering and Informatics, 10(2). <https://doi.org/10.11591/eei.v10i2.2810>
- Shah, A. A., Chowdhry, B. S., Memon, T. D., Kalwar, I. H., & Andrew Ware, J. (2020). Real time identification of railway track surface faults using canny edge detector and 2D discrete wavelet transform. Annals of Emerging Technologies in Computing, 4(2). <https://doi.org/10.33166/AETiC.2020.02.005>
- The Mathworks, Inc. MATLAB, Version 9.6, 2019. (2019). MATLAB 2019b - MathWorks. In www.Mathworks.Com/Products/Matlab.

Capítulo X - Aplicaciones con métodos combinados de Inteligencia Artificial

En este capítulo se muestran aplicaciones avanzadas con métodos combinados. El objetivo es comparar diferentes técnicas como PCA, LDA, diferentes clasificadores y redes neuronales.

En el primer ejemplo (A) se clasifican muestras de vinos con Python. Se utilizan técnicas de Análisis Multivariado de datos para reducir dimensión de los datos de entrada y luego se aplican diferentes clasificadores y redes neuronales.

En el segundo ejemplo (B) se clasifican muestras de flores del conjunto de datos IRIS mediante el software Orange. Este software utiliza bloques sin necesidad de escribir código de programa. Es una herramienta simple de usar que contiene muchos métodos, pero se debe utilizar con cuidado, hay que comprender las técnicas utilizadas y modificar los parámetros utilizados.

En el tercer ejemplo (C) se muestra una aplicación de Espectroscopía de Plasma Inducida por Láser (LIBS), se procesan datos de mediciones y se muestran los resultados. El propósito de la espectroscopía de emisión atómica es determinar la composición elemental de la muestra. Se vaporiza y/o atomiza la muestra para producir especies atómicas libres (neutras o iónicas) que son excitadas para luego detectar la luz que emiten y finalmente analizarla.

En el último ejemplo (D), se generan datos en forma aleatoria y se comparan diferentes métodos de inteligencia artificial con Python y RapidMiner. Siendo RapidMiner una herramienta muy potente para minería de datos y aprendizaje automático, trabaja con diagrama en bloques y posee muchos modelos simples de implementar.

A) Clasificación de vinos. Ejemplo de combinación de Análisis Multivariado de datos, clasificadores y redes neuronales con software Python

Ejercicio 10.1

Ejemplo de clasificación de muestras de vinos, mediante PCA y LDA, clasificadores y redes neuronales.

En este ejemplo se utilizan técnicas de Análisis Multivariado de datos: PCA y LDA en combinación con diferentes clasificadores y redes neuronales para clasificar vinos (Scikit-learn, 2021), (Webb, 2011).

Luego de realizar PCA y LDA, utilizamos 3 conjuntos de datos que llamamos X, X_r1 y X_r2 :

X: datos originales de dimensión 178 x 13 (sin PCA, ni LDA)

X_r1: datos reconstruidos con PCA de dimensión 178 x 2

X_r2: datos reconstruidos con LDA, dimensión 178 x 2

Para cada conjunto aplicamos los siguientes clasificadores:

'KNN': Clasificador KNN

'LR': Regresión Logística

'NB': Gaussiana

'CART': Clasificador basado en árbol de decisión

'SVM': basado en Máquinas de soporte de vectores (SVM)

'MLP_01': Red Neuronal Perceptron Multi Capa, topología 1

'MLP_02': Red Neuronal Perceptron Multi Capa, topología 2

Los datos de en entrenamiento y test se deben separar al inicio del programa.

Para facilitar la comprensión del programa completo, a continuación, se muestra un programa simplificado

```
import matplotlib.pyplot as plt ##
from sklearn import datasets ##
from sklearn.decomposition import PCA ##
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis ##
import numpy as np ##
##### Modelos de clasificación
from sklearn import model_selection # Selección y evaluación de modelos ##
from sklearn.linear_model import LogisticRegression ### Regresión Logística
from sklearn.tree import DecisionTreeClassifier ### Clasificador con árbol de decisión.
from sklearn.neighbors import KNeighborsClassifier ## Clasificador KNN ##
from sklearn.naive_bayes import GaussianNB # Bayes gaussiano
from sklearn.svm import SVC # C-Support Vector Classification.
from sklearn.model_selection import train_test_split # Divide matrices en prueba y test aleatorio
from sklearn.neural_network import MLPClassifier # red Muli Layer Perceptron
plt.rc('font', size=15)    # Tamaño de fuente (default)
##### Importamos datos
""" https://scikit-learn.org/stable/datasets/toy_dataset.html
load_iris , load_boston regression ,load_iris classification
load_diabetes regression , load_digits classification
load_linnerud the physical excercise ,load_wine classification
load_breast_cancer breast cancer wisconsin classification """
datos01 = datasets.load_wine()
X = datos01.data ;y = datos01.target ##
target_names = datos01.target_names
l1, l2 = np.size(X,0) , np.size(X,1) ; print("Tamaño de X: ", l1,l2)
# Opcional: ruido = np.random.randn(178,13) * 1 ; X = X + ruido ;
##### Análisis PCA
```

```

pca = PCA(n_components=2)
X_r1 = pca.fit(X).transform(X) # X_r1: datos reconstruidos con PCA
l1, l2 = np.size(X_r1,0) , np.size(X_r1,1)
print("Tamaño de X_r1: ", l1,l2)
print('Porcentaje de varianza (primeras 2 componentes): %s'
      % str(pca.explained_variance_ratio_) , "\n")
# Graficamos PCA
plt.figure(figsize=(8,6))
colors = ['red', 'turquoise', 'blue']
marcadores = ['^', 'o', 's']
lw = 2 ##
for color, i, target_name , marcador in zip( colors, [ 0, 1, 2 ] , target_names, marcadores ):
    plt.scatter( X_r1[y == i, 0] , X_r1[ y == i, 1 ] , color=color, alpha = .8,
                lw= lw, marker= marcador, label= target_name)
plt.legend( loc = 'best' , shadow = False, scatterpoints = 1 ) ## 
plt.title('PCA de Vinos')
str1 ='PC 1: %0.4f' %pca.explained_variance_ratio_[0] + '%'
str2 ='PC 2: %0.3f' %pca.explained_variance_ratio_[1] + '%'
plt.xlabel(str1) ; plt.ylabel(str2)
##### Análisis LDA #
Ida = LinearDiscriminantAnalysis( n_components=2 ) ##
X_r2 = Ida.fit( X , y ).transform( X ) # X_r2: datos reconstruidos con LDA
l1, l2 = np.size(X_r2,0) , np.size(X_r2,1)
print("Tamaño de X_r2: ", l1,l2, "\n" )
# Graficamos LDA
plt.figure(figsize=(8,6))
for color, i, target_name , marcador in zip( colors, [0, 1, 2], target_names, marcadores ):
    plt.scatter( X_r2[ y == i, 0 ] , X_r2[ y == i, 1 ] , color = color, alpha=.8,
                lw=lw, marker= marcador, label= target_name)
plt.legend( loc = 'best' , shadow =False, scatterpoints= 1 ) ##
plt.title('LDA de Vinos') #
plt.xlabel('Componente 1') ; plt.ylabel('Componente 2') ; plt.show()
##### Definimos los modelos y clasificamos
conjunto_modelos = []
conjunto_modelos.append(('KNN', KNeighborsClassifier()))
conjunto_modelos.append(('LR', LogisticRegression(max_iter=2500)))
#conjunto_modelos.append(('LDA', LinearDiscriminantAnalysis()))
conjunto_modelos.append(('NB', GaussianNB()))
conjunto_modelos.append(('CART', DecisionTreeClassifier()))
conjunto_modelos.append(('SVM', SVC()))
conjunto_modelos.append(('MLP_01', MLPClassifier(solver ='lbfgs', alpha =1e-5,
                                               hidden_layer_sizes=(10, 3), random_state=1, max_iter=2000) ))
conjunto_modelos.append(('MLP_02',
                         MLPClassifier( hidden_layer_sizes = ( 32, 16 ),
                         activation = "relu", solver = "adam", learning_rate = "constant",
                         learning_rate_init = 0.02, # tasa de aprendizaje inicial
                         early_stopping=True, max_iter = 3000) ))
##### Preparamos datos
# Definimos X: datos y separamos en entrenamiento y test
indices = np.arange(l1)

```

```

# X = X # (sin PCA, ni LDA)
X_entren0, X_test0, y_entren0, y_test0 , idx1, idx2 = train_test_split(X, y, indices, test_size=0.25)
#X_entren0, X_test0, y_entren0, y_test0 = train_test_split(X, y, test_size=0.25)
# X = X_r1 # PCA
X_entren1=X_r1[idx1] ; X_test1=X_r1[idx2] ; y_entren1=y[idx1]; y_test1=[idx2]
# X = X_r2 # LDA
X_entren2= X_r2[idx1] ; X_test2=X_r2[idx2] ; y_entren2=y[idx1]; y_test2=[idx2]
# Armamos conjunto de datos sin PCA, ni LDA, con PCA y con LDA
X_entren =[X_entren0, X_entren1, X_entren2]
X_test =[X_test0, X_test1,X_test2]
y_entren =[y_entren0, y_entren1, y_entren2]
y_test =[y_test0, y_test1, y_test2]
##### Clasificamos con distintos modelos
resultados = [],[],[] ;nombres = [],[],[] ; errores = [],[],[] ; seed = 10 ; scor = 'accuracy'
i=[0, 1, 2]
for X_entren, X_test, y_entren, y_test,i in zip(X_entren, X_test, y_entren, y_test,i):
    for nombre_clasif, modelo in conjunto_modelos:
        kfold = model_selection.KFold(n_splits=7, random_state=seed, shuffle=True)
        cv_resultados = model_selection.cross_val_score(modelo, X_entren, y_entren,
            cv=kfold, scoring=scor)
        resultados[i].append(cv_resultados.mean())
        errores[i].append(cv_resultados.std())
        nombres[i].append(nombre_clasif)
        texto = "%s: %f (%f)" % (nombre_clasif, cv_resultados.mean(), cv_resultados.std())
        print(texto)
##### Graficos Bar
ancho_barra = 0.25 # ancho de barra
fig = plt.subplots(figsize =(12, 8))
# Posición en el eje x
largo1 = len(resultados[0])
pos_x1 = np.arange( largo1 )
pos_x2 = [x + ancho_barra for x in pos_x1]
pos_x3 = [x + ancho_barra for x in pos_x2]
plt.bar(pos_x1, resultados[0], yerr=errores[0], capsize=5, color ='r', hatch="/",
        width = ancho_barra, edgecolor ='grey', label ='sin PCA, ni LDA')
plt.bar(pos_x2, resultados[1], yerr=errores[1], capsize=5, color ='g', hatch='\\',
        width = ancho_barra, edgecolor ='grey', label ='PCA')
plt.bar(pos_x3, resultados[2], yerr=errores[2], capsize=5, color ='b',
        width = ancho_barra, edgecolor ='grey', label ='LDA')
plt.xlabel('Modelos', fontweight ='bold', fontsize = 17)
plt.ylabel('Precisión', fontweight ='bold', fontsize = 17)
# Agregamos Xticks
plt.xticks([r + ancho_barra for r in range(largo1)], nombres[0])
plt.tight_layout() # plt.ylim(0, 1.23)
plt.legend() ; plt.grid(color='g', linestyle='--', linewidth=0.2)
plt.savefig('Figura 3.png') ; plt.show()

```

Resultados

En las Fig. 157 se muestran los resultados PCA y LDA, donde se proyectan los datos en 2 dimensiones. Se obtiene mejor separación de las distintas clases para el caso de LDA.

Posteriormente se toman los datos originales (X), los datos reconstruidos con PCA y los datos reconstruidos con LDA y se clasifican con distintos algoritmos. En la Fig. 158 se muestran los resultados de precisión en la clasificación para las diferentes combinaciones de algoritmos.

En este ejemplo no se observan buenos resultados en el uso de PCA, aunque si se cambian los datos se pueden obtener mejores resultados PCA respecto de LDA. Se observa buena clasificación en los métodos combinados con LDA, ver Fig. 157 y Fig. 158.

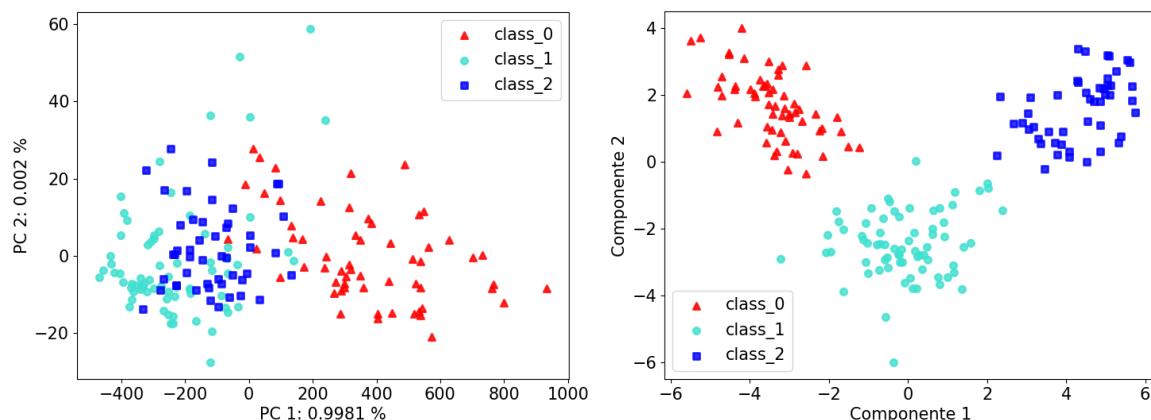


Fig. 157 – Resultados PCA y LDA

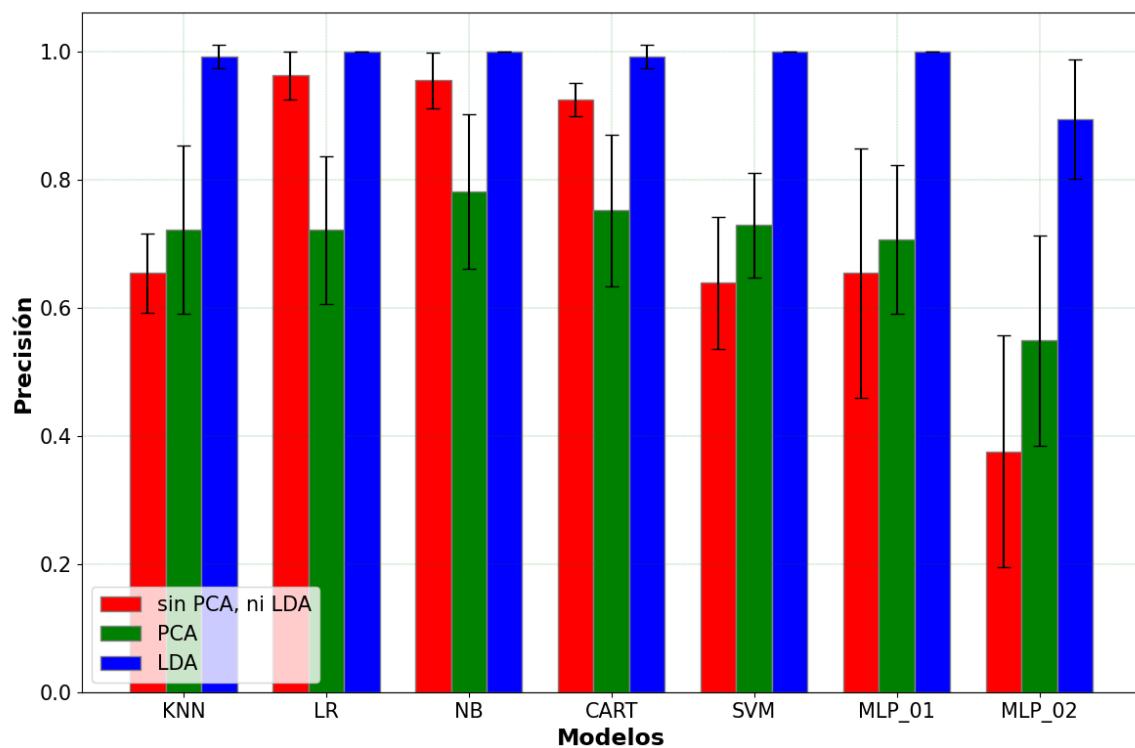


Fig. 158 – Resultados de Precisión en la clasificación de vinos para métodos combinados. Se utilizan datos originales, datos proyectados con PCA y con LDA combinados con 7 clasificadores.

B) Ejercicio de Algoritmos combinados con software Orange y conjunto de datos Iris

Orange es un software integral basado en componentes para el aprendizaje automático y la minería de datos, desarrollado en el Laboratorio de Bioinformática de la Facultad de Ciencias de la Información y la Computación, Universidad de Ljubljana, Eslovenia. Es de código abierto y presenta una interfaz para armar diagrama en bloques (Demsar, 2013), (Anggraini, 2019).

Ejercicio 10.2

Ejemplo de separación de muestras de flores del conjunto de datos IRIS mediante Orange

Se desea clasificar el conjunto de datos de flores IRIS que se encuentra disponible en las carpetas de la instalación de Orange (Demsar, 2013), (Vaishnav, 2018), (Wiguna, 2021). No se utilizan las imágenes, sino que se usan los anchos y largos de los pétalos y de los sépalos, comprende 3 categorías distintas como se observa en la Fig. 159.

Se pide:

Abrir software Orange, implementar el diagrama de la Fig. 160 y cargar el conjunto de datos IRIS.

Configurar todos los bloques. Analizar el funcionamiento y los resultados obtenidos



Iris Setosa



Iris versicolor



Iris Virginica

Fig. 159 – Conjunto de datos de 4 variables correspondiente a 3 tipos de flores Iris.

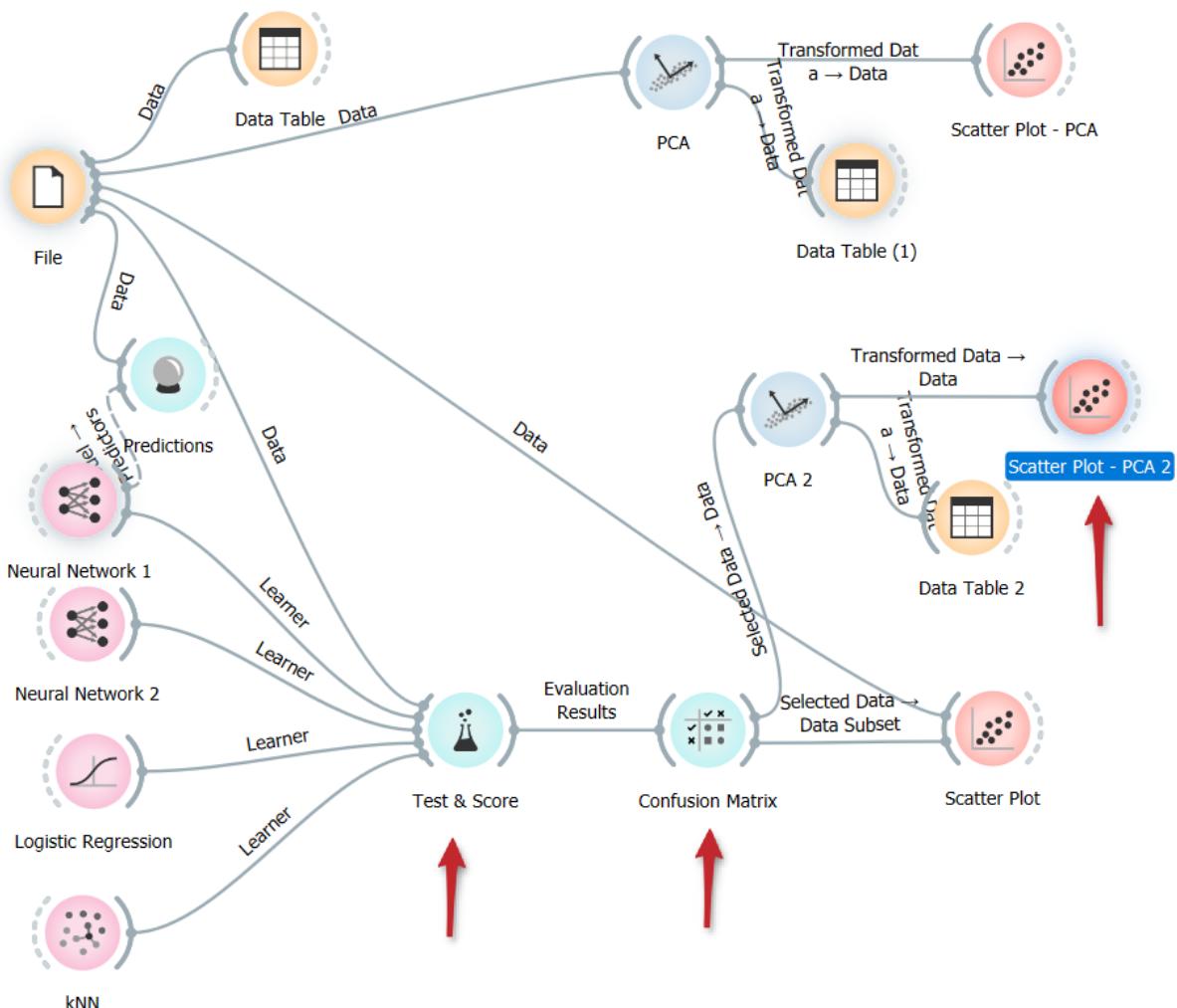


Fig. 160 – Diagrama en software Orange para análisis del conjunto de datos IRIS.

Resultados

Los resultados obtenidos correspondientes a las matrices de confusión se observan en las Fig. 161 y Fig. 162.

Resulta importante notar que una mala configuración de la red neuronal puede generar muchas clasificaciones erróneas. En este ejemplo se obtienen resultados satisfactorios para la red neuronal uno (1).

		Predicted					Predicted				
		Iris-setosa	Iris-versicolor	Iris-virginica			Iris-setosa	Iris-versicolor	Iris-virginica		
Actual	Iris-setosa	50	0	0	Σ	50	50	0	0	Σ	50
	Iris-versicolor	0	46	4		50	0	47	3		50
	Iris-virginica	0	3	47	Σ	50	50	49	51	Σ	150

Fig. 161 – Matrices de confusión para: izq.) KNN, der) Regresión Logística

Predicted			Predicted					
	Iris-setosa	Iris-versicolor	Iris-virginica		Iris-setosa	Iris-versicolor	Iris-virginica	
Actual	49	1	0	Σ	50	0	0	Σ
Iris-setosa	49	1	0	Σ	50	0	0	Σ
Iris-versicolor	0	44	6	Σ	50	20	18	50
Iris-virginica	0	3	47	Σ	50	0	0	50
Σ	49	48	53	Σ	70	18	62	Σ

Fig. 162 – Matrices de confusión para: izq.) Red Neuronal 1, der) Red Neuronal 2

En la Fig. 163 se observan los resultados PCA para el clasificador KNN. Se clasifican correctamente la mayoría de los datos, se observan tres categorías.

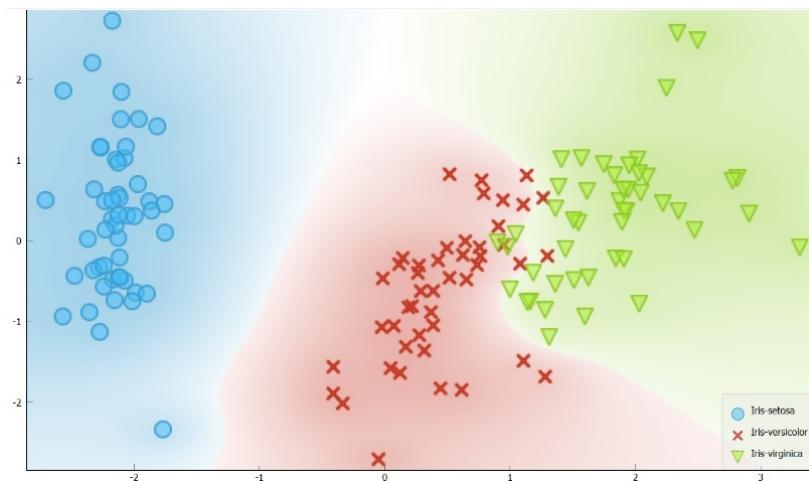


Fig. 163 – Resultados de clasificación de datos IRIS con combinación de métodos KNN y PCA

C) Algoritmos combinados para aplicaciones de Espectroscopia de Plasma Inducida por Láser

Introducción a mediciones LIBS

La Espectroscopia de Plasma Inducida por Láser, conocida como LIBS (Laser Induced Breakdown Spectroscopy) es una técnica muy novedosa de identificación, análisis y medición de materiales, donde se induce el plasma por láser. Es una técnica que permite irradiar muestras de material en cualquier de los posibles estados de condensación sólido, líquido o gaseoso y ofrece muchas soluciones en aplicaciones de diversas áreas de investigación tales como alimentos, estudios forenses, minería y restauración de obras de arte, análisis químicos, entre otras (Baudelet, 2013), (Caridi, 2017).

El propósito de la espectroscopia de emisión atómica es determinar la composición elemental de la muestra, el análisis puede ser cualitativo o también cuantitativo. El mismo consiste principalmente en la vaporización y/o atomización de la muestra para producir especies atómicas libres (neutrales o iónicas) que son excitadas para luego detectar la luz que emiten y finalmente analizarla (Beccaglia, 2006), (Manzoor, 2014), (Custo, 2005), (Massacane, 2010), (Doucet, 2007).

LIBS, también llamada espectroscopía de ablación láser (LAS), perteneciente a la espectroscopía de emisión atómica, ha sido considerada como una técnica de alto impacto para el análisis químico verde debido a sus características relevantes, tales como tiempo de análisis rápido, detección de elementos múltiples en cualquier tipo de material (sólido, líquido y gaseoso), alta resolución espacial (en la gama μm) y alto potencial para realizar análisis *in situ* o a distancia. Basándose en las capacidades únicas, la técnica LIBS ha sido testigo de un enorme crecimiento y ha sido ampliamente aplicada en diversos campos, como el monitoreo ambiental, las investigaciones arqueológicas, las aplicaciones geológicas, la detección biomédica, el análisis industrial, la agricultura, y la exploración espacial (Baudelet, 2013), (Caridi, 2017), (Lemmon, 2017).

Los elementos metálicos emiten en la zona del espectro UV- Visible que permite identificarlos por medio de LIBS para lo cual se debe tener un sistema suficientemente sensible.

Se realizan mediciones de pequeñas muestras de suelos y de uranio con la tecnología denominada LIBS (Laser-Induced Breakdown Spectroscopy o Espectroscopía de Ablación Inducida por Láser). En la Fig. 164 se muestra un esquema básico de medición LIBS armado en las instalaciones de la Comisión Nacional de Energía Atómica. Para el análisis se utilizó un espectrómetro LIBS 2500 de Ocean Optics. El análisis se realiza en el rango de 200 nm a 700 nm. Se utiliza un Láser Nd: YAG con $\lambda=1064$ nm y una lente de distancia focal de 6 cm. Para establecer las condiciones de máxima intensidad se utilizan 3 fluencias de salida: 20, 32 y 64 mJ / cm^2 resultando óptima esta última para la adquisición de los espectros (Vázquez, 2010), (NIST, 2021).

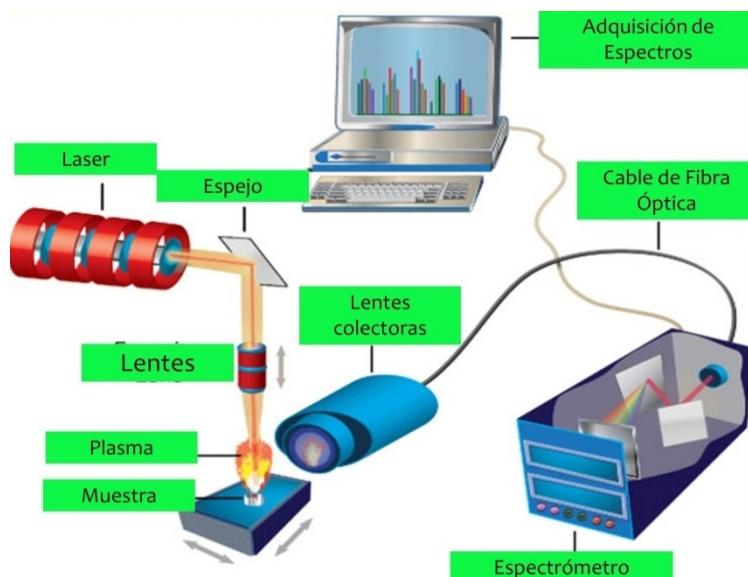


Fig. 164 – Esquema básico de medición LIBS

Palabras claves: Procesamiento de Señales; Análisis Quimiométrico; Algoritmos; Reconocimiento de Patrones; LIBS; Espectroscopía; Suelos; Medio Ambiente

Algoritmos de reconocimiento de patrones aplicados a LIBS

El procesamiento avanzado de datos de mediciones de LIBS (Espectroscopia de Plasma Inducida por Láser) resulta de gran interés para el análisis de suelos. El análisis de señales junto con el reconocimiento de patrones y técnicas quimiométricas, permite la clasificación y cuantificación de diferentes analitos. Sin embargo, los diferentes ajustes de LIBS afectan significativamente las señales medidas. A su vez, existe una amplia variedad de técnicas quimiométricas a utilizar. Todo esto genera un volumen grande de datos y resultados que deben compararse entre sí, para determinar los métodos óptimos. Cabe señalar que se pueden implementar numerosos métodos de procesamiento de datos. Este procesamiento y su correspondiente modificación de algoritmos, si se realiza manualmente resulta muy tedioso, consume mucho tiempo y requiere personal altamente capacitado. Por ello, se implementó un software propietario con algoritmos avanzados para el procesamiento de datos, que simplifica enormemente las tareas de análisis, mejora las respuestas obtenidas y es simple de usar para personal no capacitado en programación de algoritmos.

Este software de reconocimiento de patrones fue programado en Python, y en Orange. Este entorno se utilizó para automatizar la carga de datos y realizar preprocesamiento de señales, análisis quimiométrico, presentación de resultados y comparación de métodos. Los algoritmos se construyeron utilizando técnicas de reconocimiento de patrones basadas en análisis de datos multivariados, análisis de componentes principales (PCA), análisis estadístico, funciones discriminantes, inteligencia computacional y redes neuronales. Como resultado de este trabajo, se obtuvo una herramienta de reconocimiento de patrones orientada a la medición LIBS fácil de usar, que es capaz de comparar diferentes métodos de procesamiento de datos.

El análisis elemental de suelos utilizando técnicas LIBS en conjunto con técnicas quimiométricas, permite clasificar y cuantificar distintos analitos.

En la actualidad, el método de curva de calibración se utiliza ampliamente para la medición de las concentraciones en el análisis de LIBS (Mukhono,2013). Recientemente, los métodos quimiométricos multivariados combinados con la tecnología LIBS mostraron un fuerte crecimiento en el campo del análisis de suelos. Para determinar el contenido elemental, los métodos de análisis multivariados, como la regresión parcial de cuadrados mínimos (PLSR), la red neuronal artificial (ANN), la máquina de soporte vectorial (SVM), arboles de regresión aleatoria (RFR), la regresión multi lineal (MLR), la regresión de componentes principales (PCR), se aplicaron ampliamente a los datos LIBS (Feng,2011), (Feng,2013), (Zhang,2015).

Por otro lado, los investigadores desarrollaron una gran variedad de maneras de lidiar con los distintos métodos de discriminación de materiales. En general, la identificación de las líneas de emisión de un elemento significativo o el uso de múltiples líneas de emisión LIBS (cálculo de las relaciones de intensidad de las líneas seleccionadas) fueron los dos métodos para clasificar las muestras de suelos. Además, los análisis multivariados ya se han aplicado ampliamente a los datos de LIBS con fines de clasificación, como la clasificación de muestras de escoria mediante el análisis discriminante de cuadrados mínimos parciales, discriminación de rocas sedimentarias con métodos SVM, métodos discriminantes PLS (PLS-DA) para distinguir características de muestras geológicas, PCA en datos de suelos contaminados usando un sistema móvil LIBS, identificación de rocas utilizando tres métodos: PCA, PLS-DA y software de modelado independiente de analogía de clases (SIMCA), PCA para clasificación de plásticos, ajuste de las ponderaciones espectrales (ASW) para la identificación de polímeros, clasificación de rocas mediante LIBS remotos utilizando análisis de componentes independientes (ICA), análisis de agrupamiento jerárquico (HCA) para clasificar el

tejido de pollo (cerebro, pulmón, bazo, hígado, riñón y músculo esquelético), clasificación de los comprimidos farmacéuticos basados en SIMCA, clasificación de juguetes con posibles elementos tóxicos usando método de los k-vecinos más cercanos (KNN, por sus siglas en inglés). Sin embargo, se han reportado pocos estudios sobre clasificación de variedades de suelo utilizando LIBS junto con análisis multivariados (Ducet,2013), (Martín,2005), (Zhu,2014).

Existen diferentes programas de análisis quimiométrico, a modo de ejemplo se pueden utilizar estos programas “The Unscrambler X10.1” (CAMO PROCESS AS, Oslo, Norway), MATLAB R2019a software (TheMathWorks, Inc., Natick, MA, USA, 2019), Origin Pro 8.0 SRO (Origin Lab Corporation, Northampton, MA, USA), Microsoft Excel y Chemoface. Sin embargo, para obtener resultado final, se deben combinar programas de distintos fabricantes con formatos incompatibles, con muchas limitaciones y fallas, no se adecuan correctamente al problema de interés, se desconoce el funcionamiento interno y no se pueden modificar los parámetros internos de procesamiento. Por tal motivo se propone implementar un software con algoritmos avanzados de procesamiento de datos que simplifique notablemente las tareas de procesamiento, mejore las respuestas obtenidas y sea de uso fácil para personal no capacitado en manejo de datos.

El desarrollo de metodologías más simples y de bajo costo para la medición de diversos analitos en matrices de base metálica, puede resultar en un mayor y mejor control de contaminación de suelos y aguas, evitando el aumento de los efectos nocivos en la salud de las poblaciones afectadas.

Ejemplo de Mediciones de suelos con técnicas LIBS

El suelo tiene una composición química extremadamente compleja y muy diversa (Martín, 2013), ya que contiene muchos componentes como minerales, materia orgánica, organismos vivos, fósiles, aire y agua. También incluye muchas clases de compuestos orgánicos que abarcan un amplio rango de pesos moleculares e incluyen carbohidratos, aromáticos, almidones, compuestos que contienen nitrógeno y ácidos grasos (Yu,2016). Las propiedades físicas, biológicas y químicas de los suelos podrían cambiar significativamente como resultado de actividades humanas tales como la vivienda y la agricultura. Teniendo en cuenta la diversidad de los contenidos, la calidad y el uso de los suelos, resulta de gran importancia un estudio científico sistemático sobre la composición y el tipo de elementos que lo conforman (Mizolek, 2006). La detección de la abundancia o deficiencia de los elementos del suelo y la identificación de los tipos de suelos son los puntos clave de las herramientas de adquisición de información en la agricultura de precisión y también proporciona una base teórica para prevención del suelo contaminado por metales pesados y desarrollo sostenible de la agricultura (Doucet,2007).

Convencionalmente, la discriminación de los tipos de suelo dependía principalmente de la observación de las características geomorfológicas (color, tamaño de grano, aspecto y otras propiedades físicas), que era determinada por personal profesional y estaba afectada por la subjetividad del hombre. Los investigadores demostraron que el estado de fertilidad de la tierra, el suelo y la productividad de sus cultivos podrían considerarse como una base para el manejo de los cultivos y la variación del suelo. El análisis químico, incluyendo espectrometría de absorción atómica (AAS), espectroscopia de fluorescencia de rayos X (XRFS), espectrometría de emisión atómica acoplada inductivamente (ICP-AES), análisis de composición elemental y otras sustancias son un método eficaz para clasificar diferentes suelos, pero de alto costo y requieren mucho tiempo.

Ejercicio 10.3

Ejemplo de separación de muestras LIBS mediante programa con interfaz gráfica de Orange

En este ejemplo se procesan los datos provenientes de mediciones LIBS mediante un programa implementado con la interfaz gráfica de Orange y se comparan distintos algoritmos asociados.

Este programa se puede utilizar en distintas áreas de la investigación, ya que, realizando los cambios pertinentes se pueden obtener resultados rápidamente. Se deberá cambiar la matriz de datos o un grupo de señales, ajustar parámetros, analizar los métodos utilizados y realizar pequeñas modificaciones. Mediante el programa se pueden analizar y comparar distintos métodos. Esto puede facilitar considerablemente el trabajo de muchos investigadores. En la Fig. 165 se muestra el procesamiento de datos con Orange.

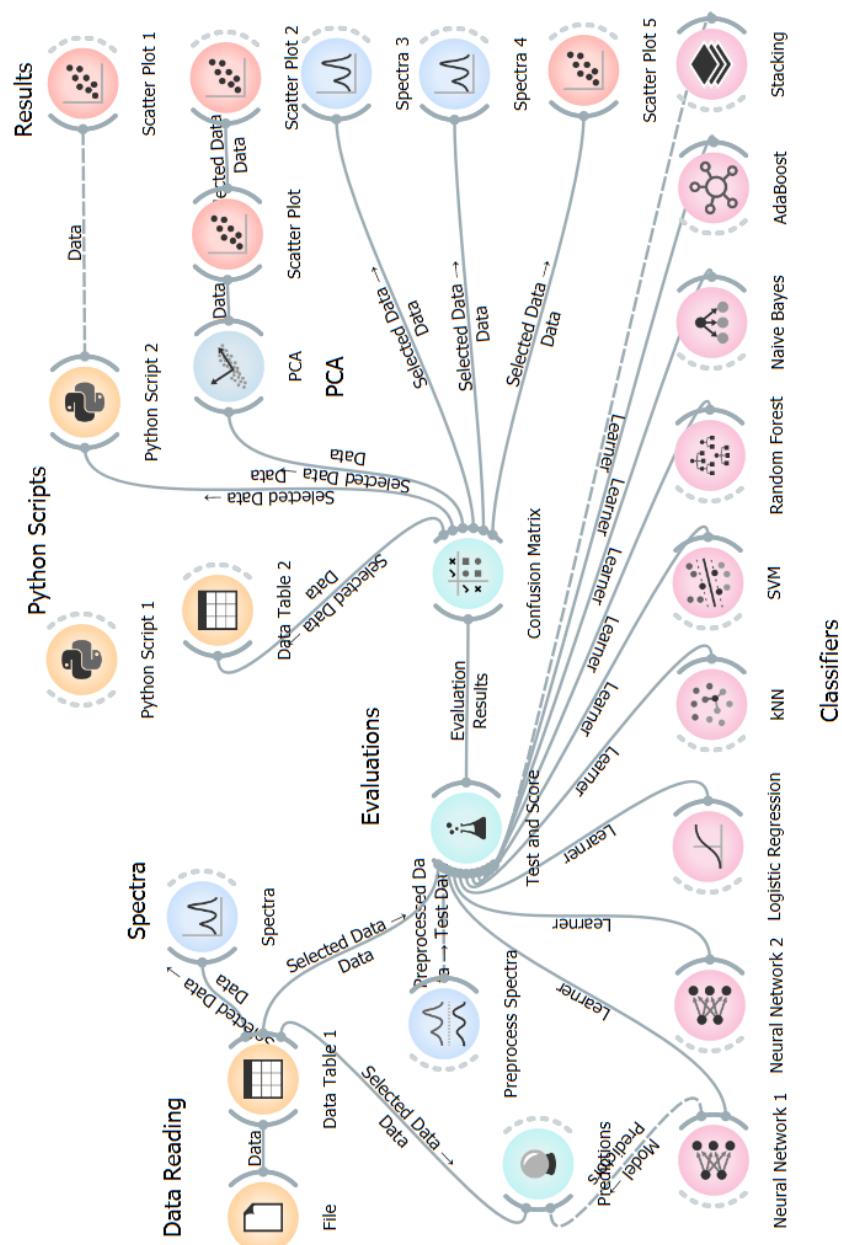


Fig. 165 – Esquema de algoritmos de procesamiento de datos LIBS con Orange

Resultados de Mediciones LIBS

En la Fig. 166 se muestra el promedio de 59 espectros para 4 concentraciones distintas de cierto elemento (información confidencial). En la Fig. 167 se pueden observar los resultados de “Test and Scores” donde se comparan todos los algoritmos. En Orange también se puede acceder fácilmente a las matrices de confusión de los algoritmos. En la Fig. 168 se muestran los resultados para la combinación de métodos PCA y redes neuronales implementado con el diagrama de la Fig. 165.

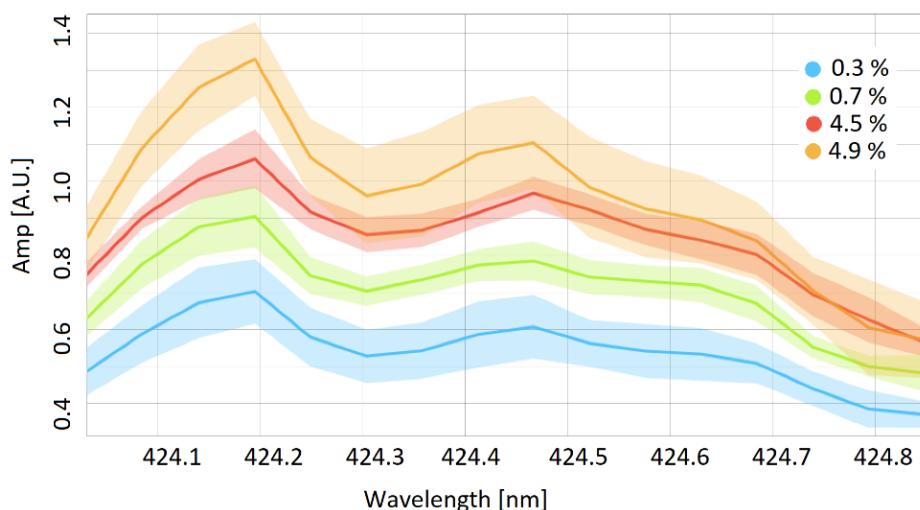


Fig. 166 – Promedio de 59 espectros, 4 grupos con distintas concentraciones

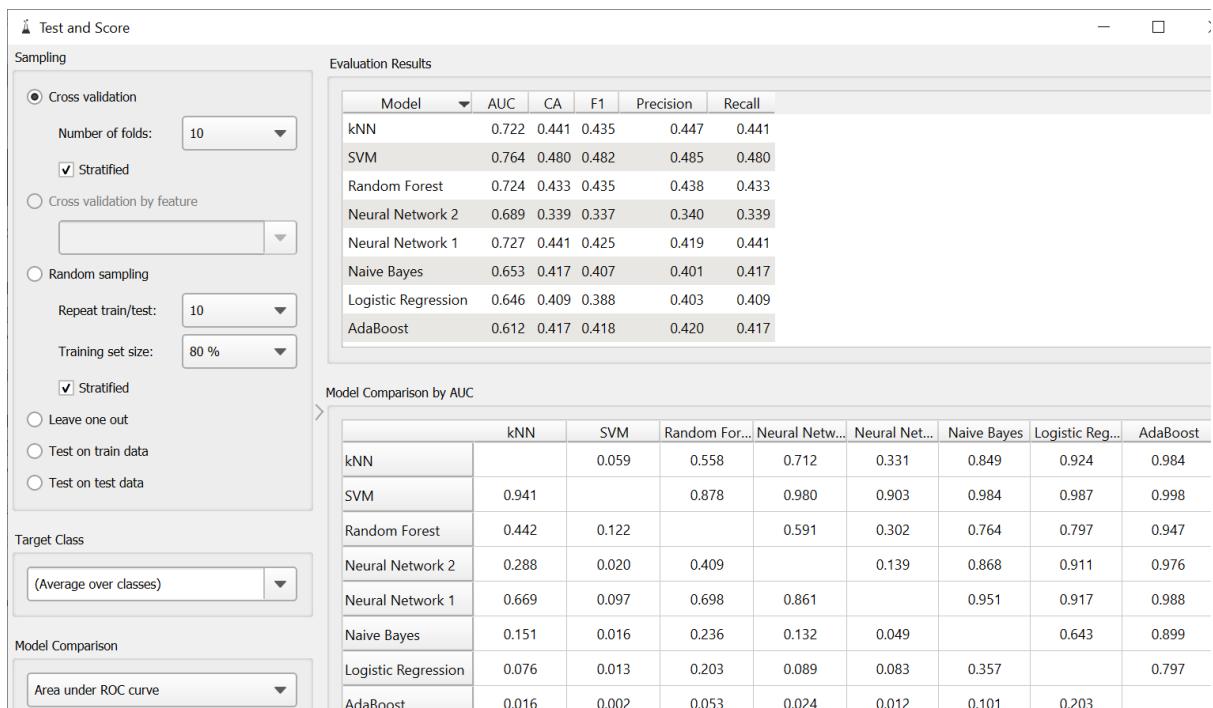


Fig. 167 – Resultados de “Test and Scores” utilizado para comparar algoritmos

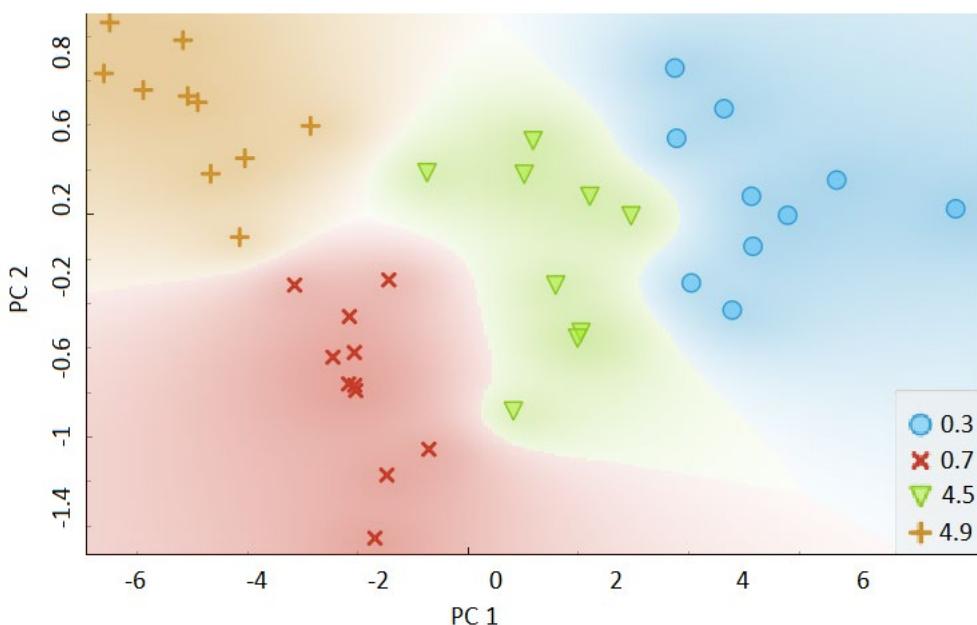


Fig. 168 – Resultados de Redes Neuronales + PCA, se observan 4 grupos correspondientes a las diferentes concentraciones.

D) Comparación de redes neuronales y diferentes técnicas de inteligencia artificial con Python y software RapidMiner

Ejercicio 10.4

Ejemplo de clasificación de muestras con diferentes redes neuronales y métodos estadísticos.

Mediante Python se genera un conjunto de datos en 2 dimensiones con 320 muestras. Se separa 40% de los datos para prueba y el resto para entrenamiento. Se utiliza el siguiente código para generar las muestras:

```
from sklearn.datasets import make_blobs
nro_muestras = 320
centros_1 = ([1, 4.2], [-1.6, 1], [-1.4, 3.7], [1.5, 1.5], [0, 7.5])
datos_1, labels = make_blobs(n_samples=nro_muestras,
                             centers=centros_1, cluster_std=0.7, random_state = 0 )
# Separamos los datos en testeo y entrenamiento
from sklearn.model_selection import train_test_split
dataset1 = train_test_split(datos_1, labels, test_size=0.4, random_state=40)
```

Luego se arma una red Perceptrón multicapa (ver ejercicio 2.5), una red tipo RBF (ver ejercicio 3.5) y una red SOM (ver ejercicio 5.2). En la Fig. 169 se muestran los resultados de clasificación con red Perceptrón.

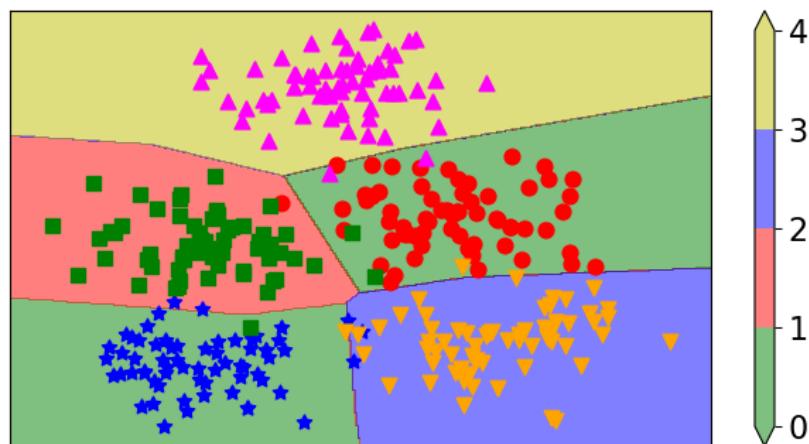


Fig. 169 – Resultados de separación mediante red neuronal Perceptrón multicapa

Se utilizan los mismos datos y se procesan con el software RapidMiner (Kotu, 2014), se utilizan 3 bloques como se observa en la Fig. 170. Donde el bloque de Python se encarga de reproducir los datos. Luego mediante la herramienta AutoModel se comparan diferentes algoritmos, ver Fig. 171, se obtienen las precisiones más altas mediante regresión logística y bosques aleatorios (Kotu, 2014).

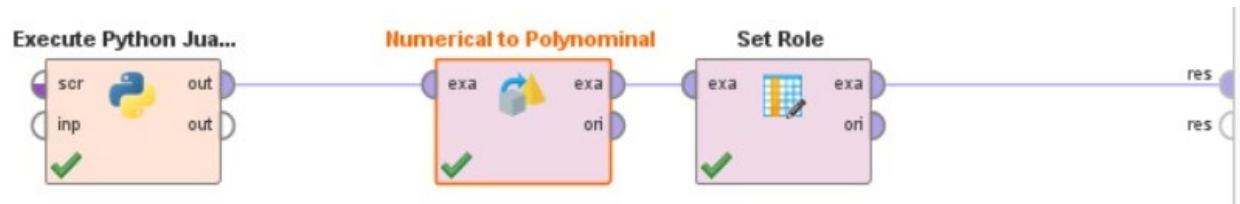


Fig. 170 – Programa implementado en RapidMiner



Fig. 171 – Resultados de precisión para diferentes métodos, implementado con RapidMiner

En la Fig. 172 se muestra el modelo generado mediante la técnica de bosques aleatorios, donde “x1” y “x2” son las variables de entrada. En la Fig. 173 se observan las curvas obtenidas para bayes ingenuo (Kotu, 2014).

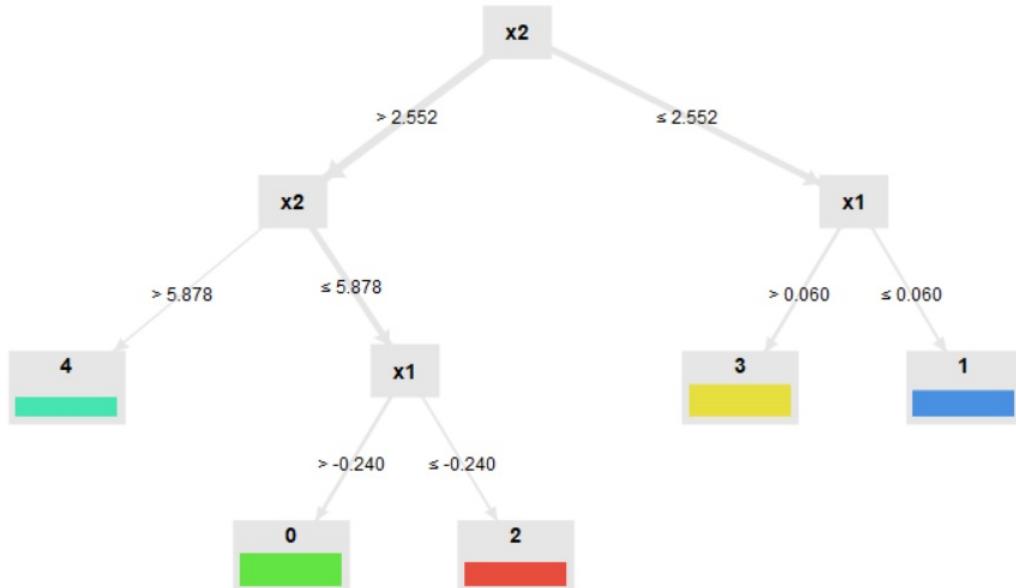


Fig. 172 – Modelo de combinación de Árboles predictivos (conocido como bosques aleatorios o en inglés: Random Forest)

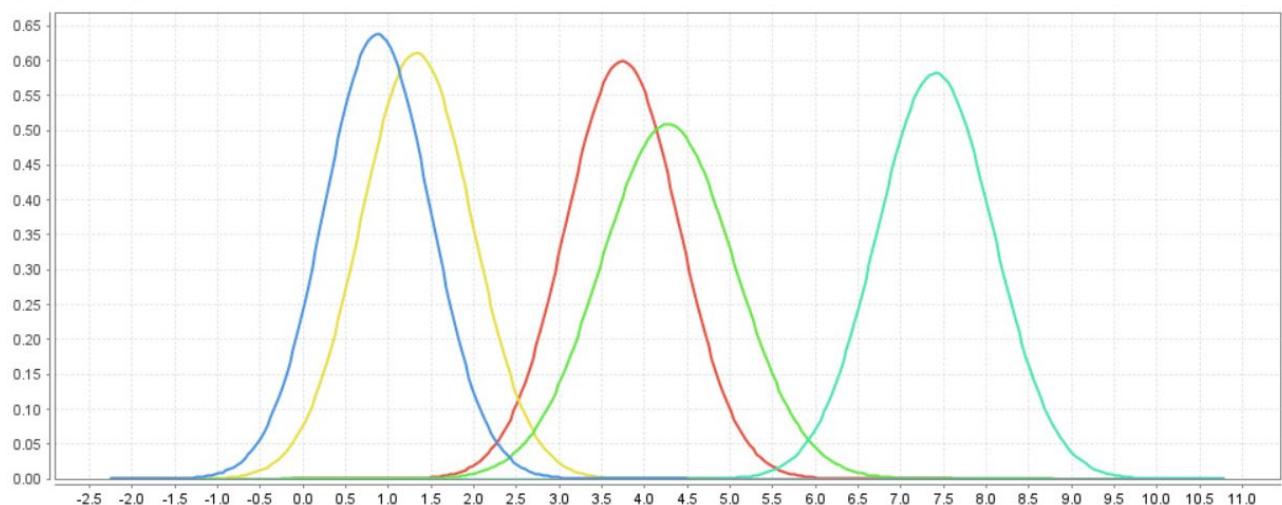


Fig. 173 – Resultado de Bayes ingenuo para la variable 2

En la Tabla XVI se muestran todos los resultados de precisión obtenidos para los diferentes modelos analizados. Se debe tener en cuenta que se utilizan modelos supervisados y no supervisados.

Tabla XVI – Comparación de diferentes modelos de redes neuronales y métodos estadísticos

Software	Modelo	Precisión para los datos de prueba %	Desviación estándar
Python	Red neuronal Perceptrón Multicapa	96,09	-
	Red neuronal RBF	96,87	-
	Red neuronal SOM	94,37	-
RapidMiner	Regresión logística	93,6	2,3
	Combinación de árboles predictivos (Random Forest)	93,5	2,6
	Máquinas de soporte vectorial (SVM)	93,5	6,1
	Bayes Ingenuo	92,3	5,0
	SVM de margen amplio (Fast Large Margin)	92,3	5,0
	Aprendizaje profundo	92,3	5,0
	Árboles de decisión	91,2	5,1
	Árboles con Potenciación del gradiente (Gradient Boosted Trees)	90,2	5,9
	Modelo lineal generalizado	83,5	10,6



Descarga de los códigos de los ejercicios

Referencias

- Anggraini, N., & Suroyo, H. (2019). Comparison of Sentiment Analysis against Digital Payment “T-cash and Go-pay” in social media Using Orange Data Mining. Journal of Information Systems and Informatics, 1(2). <https://doi.org/10.33557/journalisi.v1i2.21>
- Anggraini, N., & Suroyo, H. (2019). Perbandingan Analisis Sentimen Terhadap Digital Payment “T-cash dan Go-pay” Di Sosial Media Menggunakan Orange Data Mining. Journal of Information Systems and Informatics, 1(2).
- Baudelet, M., & Smith, B. W. (2013). The first years of laser-induced breakdown spectroscopy. In Journal of Analytical Atomic Spectrometry (Vol. 28, Issue 5). <https://doi.org/10.1039/c3ja50027f>

- Beccaglia, A. M., Rinaldi, C. A., & Ferrero, J. C. (2006). Analysis of arsenic and calcium in soil samples by laser ablation mass spectrometry. *Analytica Chimica Acta*, 579(1). <https://doi.org/10.1016/j.aca.2006.07.017>
- Caridi, F. (2017). Laser-induced breakdown spectroscopy: theory and applications, edited by Sergio Musazzi and Umberto Perini. *Contemporary Physics*, 58(3). <https://doi.org/10.1080/00107514.2017.1333528vaz>
- Custo, G., Boeykens, S., Dawidowski, L., Fox, L., Gómez, D., Luna, F., & Vázquez, C. (2005). Soil characterization by energy dispersive X-ray fluorescence: Sampling strategy for in situ analysis. *Analytical Sciences*, 21(7). <https://doi.org/10.2116/analsci.21.751>
- Demsar J, Curk T, Erjavec A, Gorup C, Hocevar T, Milutinovic M, Mozina M, Polajnar M, Toplak M, Staric A, Stajdohar M, Umek L, Zagar L, Zbontar J, Zitnik M, Zupan B (2013) Orange: Data Mining Toolbox in Python, *Journal of Machine Learning Research* 14(Aug): 2349–2353.
- Doucet, F. R., Belliveau, T. F., Fortier, J. L., & Hubert, J. (2007). Use of chemometrics and laser-induced breakdown spectroscopy for quantitative analysis of major and minor elements in aluminum alloys. *Applied Spectroscopy*, 61(3). <https://doi.org/10.1366/000370207780220813>
- Feng, J., Wang, Z., Li, L., Li, Z., & Ni, W. (2013). A nonlinearized multivariate dominant factor-based partial least squares (PLS) model for coal analysis by using laser-induced breakdown spectroscopy. *Applied Spectroscopy*, 67(3). <https://doi.org/10.1366/11-06393>
- Feng, J., Wang, Z., West, L., Li, Z., & Ni, W. (2011). A PLS model based on dominant factor for coal analysis using laser-induced breakdown spectroscopy. *Analytical and Bioanalytical Chemistry*, 400(10). <https://doi.org/10.1007/s00216-011-4865-y>
- Kotu, V., & Deshpande, B. (2014). Predictive Analytics and Data Mining: Concepts and Practice with RapidMiner. In *Predictive Analytics and Data Mining: Concepts and Practice with RapidMiner*.
- Lemmon, E. W., McLinden, M. O., & Friend, and D. G. (2017). NIST Chemistry WebBook, NIST Standard Reference Database. In *NIST Chemistry WebBook* (Issue 69).
- Massacane, A., Vorobioff, J., Pierpauli, K., Boggio, N. G., Reich, S., Rinaldi, C. A., Boselli, A., Lamagna, A., Azcárate, M. L., Codnia, J., & Manzano, F. (2010). Increasing electronic nose recognition ability by sample laser irradiation. *Sensors and Actuators, B: Chemical*, 146(2). <https://doi.org/10.1016/j.snb.2009.12.033>
- Manzoor, S., Moncayo, S., Navarro-Viloslada, F., Ayala, J. A., Izquierdo-Hornillos, R., De Villena, F. J. M., & Caceres, J. O. (2014). Rapid identification and discrimination of bacterial strains by laser induced breakdown spectroscopy and neural networks. *Talanta*, 121. <https://doi.org/10.1016/j.talanta.2013.12.057>
- Martin, M. Z., Labbé, N., Rials, T. G., & Wullschleger, S. D. (2005). Analysis of preservative-treated wood by multivariate analysis of laser-induced breakdown spectroscopy spectra. *Spectrochimica Acta - Part B Atomic Spectroscopy*, 60(7–8). <https://doi.org/10.1016/j.sab.2005.05.022>
- Martin, M. Z., Mayes, M. A., Heal, K. R., Brice, D. J., & Wullschleger, S. D. (2013). Investigation of laser-induced breakdown spectroscopy and multivariate analysis for differentiating inorganic and organic C in a variety of soils. *Spectrochimica Acta - Part B Atomic Spectroscopy*, 87. <https://doi.org/10.1016/j.sab.2013.05.026>
- Mizolek, A. W., Palleschi, V., & Schechter, I. (2006). Laser induced breakdown spectroscopy (LIBS): Fundamentals and applications. In *Laser Induced Breakdown Spectroscopy (LIBS): Fundamentals and Applications* (Vol. 9780521852746). <https://doi.org/10.1017/CBO9780511541261>

- Mukhono, P. M., Angeyo, K. H., Dehayem-Kamadjeu, A., & Kaduki, K. A. (2013). Laser induced breakdown spectroscopy and characterization of environmental matrices utilizing multivariate chemometrics. *Spectrochimica Acta - Part B Atomic Spectroscopy*, 87. <https://doi.org/10.1016/j.sab.2013.05.031>
- NIST, Datos de la Base de Datos de Referencia Estándar del NIST, Publicación: Web de Química del NIST (2021). <http://webbook.nist.gov/chemistry/>
- Scikit-learn, biblioteca de software de aprendizaje automático para Python, 2021. <https://scikit-learn.org/stable/>
- The Mathworks, Inc. MATLAB, Version 9.6, 2019. (2019). MATLAB 2019b - MathWorks. In www.Mathworks.Com/Products/Matlab.
- Vaishnav, D., & Rao, B. R. (2018). Comparison of Machine Learning Algorithms and Fruit Classification using Orange Data Mining Tool. Proceedings of the 3rd International Conference on Inventive Computation Technologies, ICICT 2018. <https://doi.org/10.1109/ICICT43934.2018.9034442>
- Vázquez, C., Custo, G., Barrio, N., Burucúa, J., Boeykens, S., & Marte, F. (2010). Inorganic pigment study of the San Pedro Gonzalez Telmo Sibyls using total reflection X-ray fluorescence. *Spectrochimica Acta - Part B Atomic Spectroscopy*, 65(9–10). <https://doi.org/10.1016/j.sab.2010.06.007>
- Wiguna, R. A. raffaidy, & Rifai, A. I. (2021). Analisis Text Clustering Masyarakat Di Twitter Mengenai Omnibus Law Menggunakan Orange Data Mining. *Journal of Information Systems and Informatics*, 3(1). <https://doi.org/10.33557/journalisi.v3i1.78>
- Webb, A. R., & Copsey, K. D. (2011). Statistical Pattern Recognition: Third Edition. In *Statistical Pattern Recognition: Third Edition*. <https://doi.org/10.1002/9781119952954>
- Yu, K. Q., Zhao, Y. R., Liu, F., & He, Y. (2016). Laser-Induced Breakdown Spectroscopy Coupled with Multivariate Chemometrics for Variety Discrimination of Soil. *Scientific Reports*, 6. <https://doi.org/10.1038/srep27574>
- Zhang, T., Wu, S., Dong, J., Wei, J., Wang, K., Tang, H., Yang, X., & Li, H. (2015). Quantitative and classification analysis of slag samples by laser induced breakdown spectroscopy (LIBS) coupled with support vector machine (SVM) and partial least square (PLS) methods. *Journal of Analytical Atomic Spectrometry*, 30(2). <https://doi.org/10.1039/c4ja00421c>
- Zhu, X., Xu, T., Lin, Q., Liang, L., Niu, G., Lai, H., Xu, M., Wang, X., Li, H., & Duan, Y. (2014). Advanced statistical analysis of laser-induced breakdown spectroscopy data to discriminate sedimentary rocks based on Czerny-Turner and Echelle spectrometers. *Spectrochimica Acta - Part B Atomic Spectroscopy*, 93. <https://doi.org/10.1016/j.sab.2014.01.001>

Capítulo XI -Aplicaciones. Procesado de señales interferométricas con redes neuronales: Estimando frecuencias

A lo largo de este capítulo se muestra el uso de redes neuronales artificiales para el análisis de señales de interferometría de baja coherencia, donde es de particular interés la determinación de frecuencias. Se estudian diferentes arquitecturas de redes neuronales, analizando su comportamiento y características principales de entrenamiento, los tiempos de entrenamiento para diferentes cantidades de datos de entrenamiento, la dependencia de la resolución en función de los datos de entrenamientos y parámetros de la red. Por último, se comparan los resultados obtenidos utilizando redes con el algoritmo tradicional de la fft. Esta innovadora forma de procesar señales abre las puertas a nuevas alternativas la hora de calcular la frecuencia de cualquier tipo de señales unidimensionales.

Introducción

Las Redes Neuronales Artificiales se han desarrollado y crecido de una manera considerable en los últimos años captando la atención no solo de la academia, sino también de disciplinas que van desde la medicina hasta las finanzas, interviniendo en problemáticas teóricas y también experimentales, que involucran a instituciones públicas como privadas y regiones locales como globales.

Siempre es bueno recordar, y volver a disfrutar, la idea que dio inicio al surgimiento de las Redes Neuronales Artificiales. ¿Alguna vez pensaron en cómo un ser humano aprende a jugar al ping pong o identificaron los pasos que va dando un recién nacido hasta que consigue hablar fluidamente? Pocas son las personas que conocen las ecuaciones de movimiento de cuerpos esféricos inmersos en fluidos, que saben estimar la trayectoria de un movimiento paraboloide en fracciones de segundo, calcular el ángulo de impacto y velocidad necesaria de la paleta para colocar la pelota justo en el ángulo opuesto de la mesa. Menos aún son los y las niñas que conocen la anatomía de las cuerdas vocales, la construcción semántica de una oración unimembre o que realizan la transformada de Fourier para identificar las componentes espectrales propias de sus seres queridos. O que memorizan las veintitantas letras del abecedario y las 2000 palabras más usadas del idioma español. Sin embargo, muchas personas juegan al ping pong y aun muchas más saben hablar. Dado que el ser humano es capaz resolver de una manera sencilla, rápida y cotidiana problemáticas que ni la tecnología más innovadora consigue y además lo consigue sin la necesidad de solucionar ninguna ecuación diferencial. ¿Por qué en vez de formular teorías ultra racionales y modelos matemáticos con tantas hipótesis y descripciones algebraicas simplemente intentamos copiarnos (o mejor aún inspirarnos) del motor central de aprendizaje humano o también llamado Cerebro? Neuronas interconectadas con otras neuronas que reciben estímulos, se activan y transmiten nuevas respuestas que serán estímulos de otras conexiones neuronales y que ante sucesivas experiencias de aprendizaje logran moderar los estímulos para conseguir el objetivo deseado. He aquí las bases fundacionales de la Inteligencia Artificial y el Machine Learning.

Estas no son técnicas nuevas, ya existen hace varias décadas, pero los avances progresivos en esta área sumado al aumento de la capacidad humana de generar, almacenar y transmitir grandes cantidades de información hicieron que estos últimos años se vuelva a realzar y valorizar su gran potencialidad. En los últimos 5 años la cantidad de publicaciones y relevancia en Inteligencia Artificial aumenta a pasos agigantados, según Google Scholar, dentro de las 10 publicaciones más citadas del año 2019 de todas las áreas más de un tercio son acerca de Redes Neuronales Artificiales. Dato que aumenta a más de un 40% para el año 2020. Además, en estos últimos años constantemente nuevas disciplinas están incorporando el uso de Inteligencia Artificial y Machine Learning. Algunas de sus finalidades son: mejorar su efectividad, automatizar procesos, detectar anomalías, reconocer comportamientos, manipular grandes cantidades de información, entre otros.

El uso de Redes Neuronales Artificiales se suma, sin lugar a duda, al paradigma ya instalado de la interdisciplinariedad. Vivimos en una sociedad con un gran nivel de desarrollo y complejidad y a un ritmo acelerado, cada área del conocimiento a calado bien profundo en sus temáticas. El desafío disruptivo actual no pasa tanto por descubrir algo nuevo si no en adaptar el conocimiento existente de una disciplina a solucionar una problemática de otra. A entrecruzar los saberes y las necesidades para desarrollar nuevos enfoques teóricos y técnicos.

Ya hace varias décadas que nos encontramos en la época de la Información, para tomar prácticamente cualquier decisión todas las áreas del conocimiento tienen que poder detectar (sea analógico, digital o de alguna otra manera), manipular, transformar y analizar datos para obtener información de valor. Tan importante es esta necesidad que, ya hace varios años, el procesamiento de señales se ha constituido en un área en sí y las redes neuronales en una de las herramientas de más desarrollo en la actualidad. Una característica muy interesante para analizar una señal es la velocidad con que se repite, es decir su frecuencia. Concepto central que nos ayuda a describir gran parte del universo que nos rodea. Nos permite explicar, por ejemplo: los movimientos oscilatorios que van desde resortes a modos vibracionales de moléculas, fenómenos ondulatorios de una ola en un estanque como de la luz viajando en el espacio, la transferencia de más de millones de símbolos por segundo a través de fibras ópticas, hasta incluso evidenciar la naturaleza cuántica del mundo de lo pequeño. Por este motivo estimar las frecuencias existentes en una señal es muy importante y hay mucho trabajo realizado sobre el tema.

La interferometría es un conjunto de técnicas que basan en el principio de interferencia para la medición de diferentes fenómenos físicos, básicamente se centra en estudiar las maneras en que se combinan diferentes ondas electromagnéticas. Son muy utilizadas en aplicaciones que van desde la astronomía hasta la física nuclear pasando por metrología y espectroscopia entre otras. El resultado de superponer dos o más ondas que interfieren entre ellas es un patrón oscilante de intensidades, donde generalmente es en la frecuencia que se codifica la información de relevancia.

Lo que se busca en este capítulo es justamente utilizar Redes Neuronales Artificiales para reemplazar herramientas convencionales del procesamiento de señales para aplicarlo en el área de interferometría de ondas electromagnéticas, puntualmente a la luz.

Interferometría de baja coherencia

La interferometría de baja coherencia (LCI por sus siglas en inglés, low coherence interferometry), es una técnica de metrología óptica que permite la medición de perfilometrías, tomografías, vibraciones, velocidades, entre otras, utilizando como elemento sensor a la luz (Huang et al., 1991) (Walecki et al., 2005). Para ello se utiliza típicamente un dispositivo conocido como interferómetro de Michelson, se puede destacar que existe una gran cantidad de interferómetros que se pueden utilizar, pero el de Michelson es el más característico. Una característica importante

es que la fuente de luz que se utiliza es de baja coherencia, esto implica que la fuente posee un ancho de banda grande (mayor a los 20 nm), si bien existen numerosas fuentes de luz que cumplen con este propósito la tecnología más usada es la de los leds super luminiscentes (SLD). En particular el sistema de detección que se utiliza es un espectrómetro que permite obtener la señal de interferencia.

En la Fig. 174 se muestra una configuración típica de un interferómetro de tipo Michelson, donde se puede ver esquemáticamente como un haz de luz proveniente de una fuente de baja coherencia se dirige al divisor de haz (beam splitter, BS), se asume que el BS divide la potencia óptica incidente de manera uniforme en brazos de muestra y de referencia, aunque muchos diseños prácticos de sistemas OCT aprovechan la división de potencia desequilibrada. A partir del BS un haz de luz se dirige a la muestra en estudio y el otro a un espejo de referencia R. El haz del brazo de muestra incide sobre un mecanismo de escaneo y óptica de enfoque que permite realizar un escaneo de la muestra, estos sistemas se controlan típicamente con una computadora. Se han desarrollado muchos sistemas de escaneo especializados para aplicaciones de imágenes de LCI en microscopía, oftalmología, endoscopía y en ensayos no destructivos como se indica en (M. R. Wang, n.d.)(Grulkowski et al., 2013) (Walecki et al., 2006).

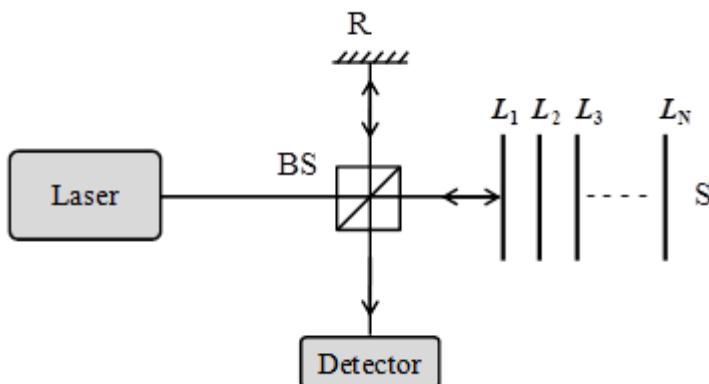


Fig. 174 – Esquema de una configuración experimental de interferencia de baja coherencia, basado en un interferómetro de Michelson

La luz reflejada de la muestra se redirige a través del mismo sistema óptico de la rama donde se encuentra la muestra, puede tener múltiples capas ($L_1,..L_N$), esto implica la capacidad de la técnica para realizar topografías y tomografías. Las reflexiones provenientes de la muestra y del espejo de referencia vuelven por el mismo camino hacia el BS, que redirige a los haces hacia el detector. Los haces después del BS se combinan generando la interferencia que se detecta con el espectrómetro. Como la luz recorre el camino tanto a la ida como a la vuelta y que además puede ser en distintos materiales, se suele usar el término diferencia de camino óptico DCO para referirse a ella. Que no es ni más ni menos que dos veces la distancia de interés multiplicado por el índice de refracción n del material $DCO = 2n(z_r - z_{L_i})$.

Se debe destacar que la cámara o array lineal en el espectrómetro detecta la irradiancia de la señal de interferencia que es función de los campos.

$$I = \frac{c}{\mu_0} \langle \mathbf{B}^2 \rangle_T \quad (11.1)$$

$$I = \epsilon_0 c \langle E^2 \rangle_T \quad (11.2)$$

Típicamente la expresión general de las señales obtenidas en el detector tiene la siguiente representación

$$\begin{aligned} i_D(k) = & \frac{\rho}{4} [s(k)(r_r + r_{s1} + r_{s2} + \dots)] \\ & + \frac{\rho}{4} \left[s(k) \sum_{n=1}^N \sqrt{r_r r_{sn}} (e^{j2k(z_r - z_{sn})} + e^{-j2k(z_r - z_{sn})}) \right] \\ & + \frac{\rho}{4} \left[s(k) \sum_{n \neq m=1}^N \sqrt{r_r r_{sm}} (e^{j2k(z_{sn} - z_{sm})} + e^{-j2k(z_{sn} - z_{sm})}) \right] \end{aligned} \quad (11.3)$$

Donde $s(k)$, es la forma espectral de la fuente de luz. En general se utilizan espectros de fuente de luz de forma Gaussiana por su conveniencia en el modelado y porque se aproxima a la forma de las fuentes de luz reales y también tiene propiedades útiles de la transformada de Fourier. La función gaussiana normalizada $s(k)$ y su transformada de Fourier inversa $S(z)$ están dadas por:

$$s(k) = \frac{1}{\Delta k \sqrt{\pi}} e^{-\left[\frac{(k-k_0)}{\Delta k}\right]^2} \leftrightarrow S(z) = e^{-z^2 \Delta k^2} \quad (11.4)$$

Aquí, k_0 representa el número de onda central del espectro de la fuente de luz y Δk representa su ancho de banda espectral, correspondiente a la mitad del ancho del espectro a 1/e de su máximo.

Al realizar la transformada de Fourier de las señales $i_D(k)$, obtenemos la señal $I_D(z)$, donde se puede apreciar las distintas características de la señal de interferencia.

$$\begin{aligned} i_D(z) = & \frac{\rho}{8} [S(z) \otimes \delta(z)(r_r + r_{s1} + r_{s2} + \dots)] \\ & + \frac{\rho}{4} \left[S(z) \otimes \sum_{n=1}^N \sqrt{r_r r_{sn}} (\delta(z \pm 2(z_r - z_{sn}))) \right] \\ & + \frac{\rho}{8} \left[S(z) \otimes \sum_{n \neq m=1}^N \sqrt{r_r r_{sm}} (\delta(z \pm 2(z_{sn} - z_{sm}))) \right] \end{aligned} \quad (11.5)$$

El primer término de la ecuación (11.5) se denomina componente "constante" o "DC". Este es el componente más grande de la corriente del detector si la reflectividad de referencia domina la reflectividad de la muestra. El segundo término de la ecuación (11.5) es la componente de correlación cruzada para cada reflexión de muestra, que depende tanto del número de onda de la fuente de luz como de la diferencia de longitud de trayectoria entre el brazo de referencia y las interfaces de la muestra donde se generan reflexiones. Este es el componente deseado para la obtención de imágenes OCT. Dado que estos componentes son proporcionales a la raíz cuadrada de las reflectividades de la muestra, normalmente son más pequeños que la componente DC. Sin embargo, la dependencia de la raíz cuadrada representa un factor de ganancia logarítmica importante sobre la detección directa de reflexiones de la muestra. El tercer término de la ecuación (11.5) son los términos de autocorrelación que representan la interferencia que ocurre entre las diferentes interfaces de muestra. Dado que los términos de autocorrelación dependen linealmente de la reflectividad de las diferentes interfaces de la muestra, serán pequeños en comparación con los otros.

Si bien las señales de interferencia $i_D(k)$ e $I_D(z)$ ilustran el principio fundamental de la LCI, en implementaciones prácticas los dispositivos de detección presentan varios factores adicionales que deben ser tomados en cuenta, como se detalla en (Dorrer et al., 2000). La señal de interferencia se obtiene mediante instrumentación que tiene limitaciones del mundo real, y normalmente se adquieren mediante una operación de muestreo para el cálculo rápido de la señal digital de su transformada de Fourier inversa.

Lo primero para tener en cuenta son aspectos relacionados a artefactos introducidos por el sistema de detección. Uno de ellos es la resolución propia del espectrómetro aquí denotada por Δk , que es la resoluciónpectral del espectrómetro (incluido el espaciado finito de los píxeles del CCD), en número de onda.

El efecto de la resoluciónpectral finita se puede modelar convolucionando la señal de interferencia ideal con una función gaussiana donde la mitad de su ancho máximo es Δk_d que interpretamos como la resoluciónpectral con el criterio ancho a mitad de altura (FWHM), que representa la función de dispersión del espectrómetro (PSF) y una función seno cardinal (sinc) que aporta el efecto del ancho finito de cada pixel, se dimensiona el ancho de cada pixel como Δk_p . Estos efectos generan una atenuación de la visibilidad de la señal a medida que se incrementa la DCO, este efecto se conoce como Fall-Off. A través de la propiedad de convolución, $i(k)$ se relaciona con el Fall-Off, en el espacio transformado la señal $I_D(z)$ se multiplica por un factor de Fall-Off, que representa una pérdida de sensibilidad para valores de z grandes y cuya forma está dada por la transformada de Fourier inversa del factor de resolución.

$$I_D(z)e^{-\left(\frac{z}{\Delta z_d}\right)^2} \sin c\left(\frac{z}{\Delta z_p}\right) \leftrightarrow i_D(k) \otimes e^{-\left(\frac{k^2}{Ra}\right)^2} \otimes \prod \left(\frac{k}{\Delta k_p}\right) \quad (11.6)$$

Donde a es el tamaño del spot de luz para cada k y R de dispersión lineal recíproca. R permite relacionar la variable k con la coordenada espacial sobre el array de pixels (eje y , distancia sobre el array CCD, tomando como origen el centro del primer pixel). En este trabajo se supone que esta relación es lineal y por lo tanto R es constante y se puede estimar como $R = \Delta k_p / \Delta y$, siendo Δy el ancho espacial del pixel. Para el caso en que el sistema de dispersión es una red de difracción el valor de R depende de la distancia focal f , la longitud focal del espectrómetro, θ , el ángulo de difracción y m el orden de difracción de la red:

$$\left. \frac{\partial k}{\partial y} \right|_{k_0} = \frac{d \cos(\theta_r) k_0^2}{2\pi m f} \quad (11.7)$$

La segunda consideración importante en el procesamiento del mundo real de datos en la técnica de LCI en el Dominio de las Frecuencias es que la detección implica muestrear la señal de interferencia $i_D(k)$, como el muestreo se realiza por medio de una cámara o array lineal la trama muestreada es finita, esto se puede representar por medio de una señal rectangular la cual posee un ancho igual al ancho de la cámara o array lineal (Δk_a) en términos de k .

$$\begin{aligned} & \left[I_D(z) e^{-\left(\frac{z}{\Delta z_d}\right)^2} \sin c\left(\frac{z}{\Delta z_p}\right) \right] \otimes \sin c\left(\frac{z}{\Delta z_a}\right) \\ & \leftrightarrow \left[i_D(k) \otimes e^{-\left(\frac{k^2}{R_a}\right)^2} \otimes \prod \left(\frac{k}{\Delta k_p} \right) \right] \prod \left(\frac{k - k_0}{\Delta k_a} \right) \end{aligned} \quad (11.8)$$

Un tercer elemento para tener en cuenta es que, la señal de interferencia $i_D(k)$ se muestrea de manera no lineal. Esto es debido al diseño del espectrómetro. Quedando entonces la señal con la siguiente forma:

$$i(k) = \left[i_D(k) \otimes e^{-\left(\frac{k^2}{R_a}\right)^2} \otimes \prod \left(\frac{k}{\Delta k_p} \right) \right] \prod \left(\frac{k - k_0}{\Delta k_a} \right) \quad (11.9)$$

$$I(z) = \left[I_D(z) e^{-\left(\frac{z}{\Delta z_d}\right)^2} \sin c\left(\frac{z}{\Delta z_p}\right) \right] \otimes \sin c\left(\frac{z}{\Delta z_a}\right) \quad (11.10)$$

En términos completos las expresiones anteriores quedan de las siguientes formas:

$$\begin{aligned}
i(k) = & \left[\frac{\rho}{4} [s(k)(r_r + r_{s1} + r_{s2} + \dots)] \dots \right. \\
& + \frac{\rho}{4} \left[s(k) \sum_{n=1}^N \sqrt{r_r r_{sn}} (e^{j2k(z_r - z_{sn})} + e^{-j2k(z_r - z_{sn})}) \right] \dots \\
& \left. + \frac{\rho}{4} \left[s(k) \sum_{n \neq m=1}^N \sqrt{r_r r_{sm}} (e^{j2k(z_{sn} - z_{sm})} + e^{-j2k(z_{sn} - z_{sm})}) \right] \dots \right] \\
& \otimes \left[e^{-\left(\frac{k^2}{Ra}\right)^2} \prod \left(\frac{k}{\Delta k_p} \right) \right] \prod \left(\frac{k - k_0}{\Delta k_a} \right)
\end{aligned} \tag{11.11}$$

$$\begin{aligned}
I(z) = & \left[\frac{\rho}{8} [S(z) \otimes \delta(z)(r_r + r_{s1} + r_{s2} + \dots)] \dots \right. \\
& + \frac{\rho}{4} \left[S(z) \otimes \sum_{n=1}^N \sqrt{r_r r_{sn}} (\delta(z \pm 2(z_r - z_{sn}))) \right] \dots \\
& \left. + \frac{\rho}{8} \left[S(z) \otimes \sum_{n \neq m=1}^N \sqrt{r_r r_{sm}} (\delta(z \pm 2(z_{sn} - z_{sm}))) \right] \dots \right] \\
& \left[e^{-\left(\frac{z}{\Delta z_d}\right)^2} \sin c \left(\frac{z}{\Delta z_p} \right) \right] \otimes \sin c \left(\frac{z}{\Delta z_a} \right)
\end{aligned} \tag{11.12}$$

De esta expresión se desprenden las características relevantes de la técnica como la resolución y el rango dinámico. La resolución dependerá claramente de la fuente de luz utilizada, que se puede expresar en términos de k o de λ .

$$Resolucion = \frac{2\sqrt{\ln(2)}}{\Delta k} = \frac{2\sqrt{\ln(2)}}{\pi} \frac{\lambda_0^2}{\Delta \lambda} \tag{11.13}$$

El rango dinámico es un poco más complejo ya que depende de la cantidad de muestras que se puedan adquirir (cantidad de pixeles), y de diferentes elementos que componen el detector (tamaños de los pixeles, red de difracción, entre lo más importantes), pero típicamente el rango de medición no supera los 3 a 5 mm.

Caso de estudio: Perfilometría/topografía

Una de las aplicaciones más interesantes son la perfilometría/topografía y la medición de distancias, la medición por LCI permite la detección de luz a alta sensibilidad, limitada solo por el ruido de disparo. Diversas aplicaciones industriales se analizan en (Dufour, 2006) (Cerrotta et al., 2015)(Morel et al., 2016). Las mediciones se pueden realizar sin contacto físico con la muestra bajo investigación. Se consigue una resolución de profundidad de unos pocos micrómetros incluso en muestras muy difusivas, mediante la supresión de la luz dispersa.

En los últimos años, la LCI se ha desarrollado vigorosamente hasta convertirse en una poderosa herramienta. La mayor complejidad y los altos volúmenes de producción en la fabricación moderna de estructuras exigen métodos de inspección, control de calidad y metrología adecuados. Las desviaciones geométricas de las estructuras y las inhomogeneidades de los materiales pueden provocar errores sistemáticos durante todos los pasos de la fabricación, lo que podría afectar la calidad y la funcionalidad de grandes volúmenes de un producto. Entre los campos de aplicación y solo por mencionar algunos de relevancia tecnológica se encuentran aplicaciones como semiconductores de potencia, energía fotovoltaica, detección de defectos en MEMS y electrónica impresa. La metrología que acompaña a la producción se vuelve vital.

En particular en el área metal mecánica, autopartista o siderúrgica, el estudio de las desviaciones no deseadas en parámetros como la rugosidad de la superficie y la topografía pueden causar problemas importantes. Esta técnica además es adecuada para ser aplicada en línea de producción.

Con la finalidad de obtener una mejor comprensión del comportamiento de las señales al realizar topografías mediante LCI, teniendo en cuenta que ahora se supone una única interfase de la muestra de interés, se reescribirán las ecuaciones (11.11) y (11.12) quedando:

$$i(k) = \frac{\rho}{4} \left[\left[s(k) \left[[(r_r + r_s)] + [\sqrt{r_r r_s} (e^{j2k(z_r - z_s)} + e^{-j2k(z_r - z_s)})] \right] \right] \prod \left(\frac{k - k_0}{\Delta k_a} \right) \right. \\ \left. \otimes \left[e^{-\left(\frac{k^2}{Ra}\right)^2} \prod \left(\frac{k}{\Delta k_p} \right) \right] \right] \quad (11.14)$$

$$I(z) = \frac{\rho}{8} \left[\left[S(z) \otimes \left[[(r_r + r_s) \delta(z)] + [\sqrt{r_r r_s} (\delta(z \pm 2(z_r - z_s))] \right] \right] \right] \\ \otimes \sin c \left(\frac{z}{\Delta z_a} \right) \left[e^{-\left(\frac{z}{\Delta z_d}\right)^2} \sin c \left(\frac{z}{\Delta z_p} \right) \right] \quad (11.15)$$

Las cuales podemos simplificar si tomamos la parte real de la señal $i(k)$, quedándonos:

$$i(k) = \frac{\rho}{4} \left[s(k) \left[[(r_r + r_s)] + [\sqrt{r_r r_s} \cos(k_2(z_r - z_s))] \right] \right] \prod \left(\frac{k - k_0}{\Delta k_a} \right) \\ \otimes \left[e^{-\left(\frac{k^2}{Ra}\right)^2} \prod \left(\frac{k}{\Delta k_p} \right) \right] \quad (11.16)$$

En la Fig. 175 se puede observar una típica señal de LCI, al realizar topografías, donde se puede apreciar como la visibilidad o pico a pico de la señal no es máximo, esto es debido a dos factores, el primero es la relación entre la reflectividad r_r y r_s , el segundo son los artefactos introducidos por el sistema de detección. Además se ve como la forma espectral de la fuente de luz utilizada genera una envolvente sobre la señal, que en este caso se utilizó la forma espectral doble gaussiana, vale aclarar que los perfiles espectrales de las fuentes de luz pueden tener diferentes formas.

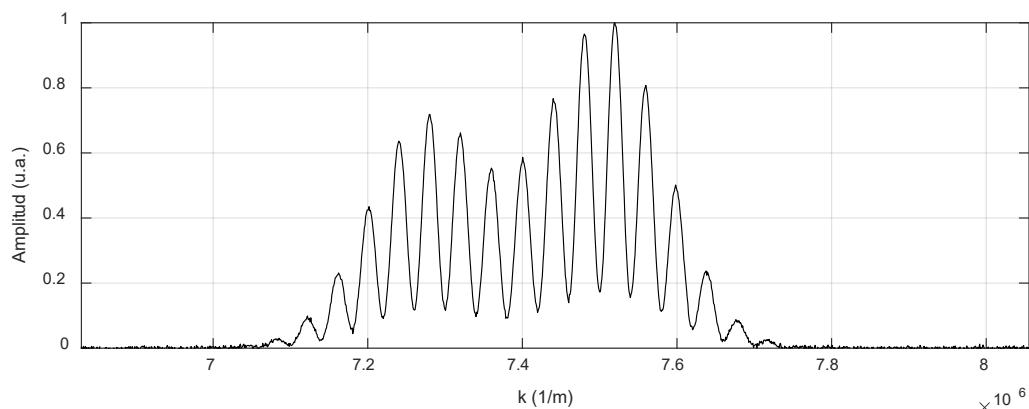


Fig. 175 – Señal típica de LCI, donde se puede percibir la oscilación de la amplitud.

Recordemos que el espectrómetro realiza un muestreo no lineal en función de k , por tal motivo después de la etapa de adquisición se realiza un remuestreo y conversión de λ a k , ver Fig. 176. La siguiente etapa realiza un filtrado de la componente de continua, luego se determina la envolvente que representa la forma espectral de la fuente de luz, luego se procede a realizar una normalización, un recorte de la señal en función de los puntos útiles, para poder realizar la transformada rápida de Fourier (fft) y finalmente la determinación de la frecuencia espacial o posición del pico, como se comenta en el artículo (Ali & Parlapalli, 2010) .

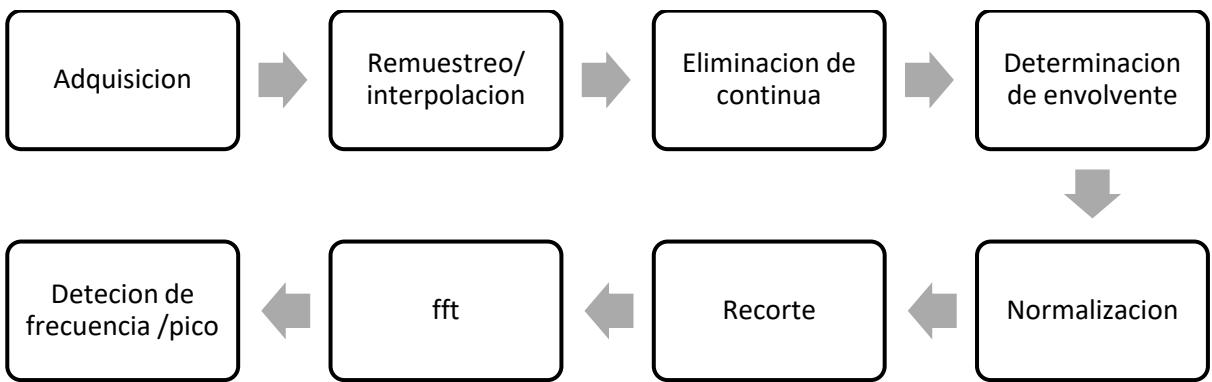


Fig. 176 – Esquema de las etapas de preprocesado convencional de una señal de LCI.

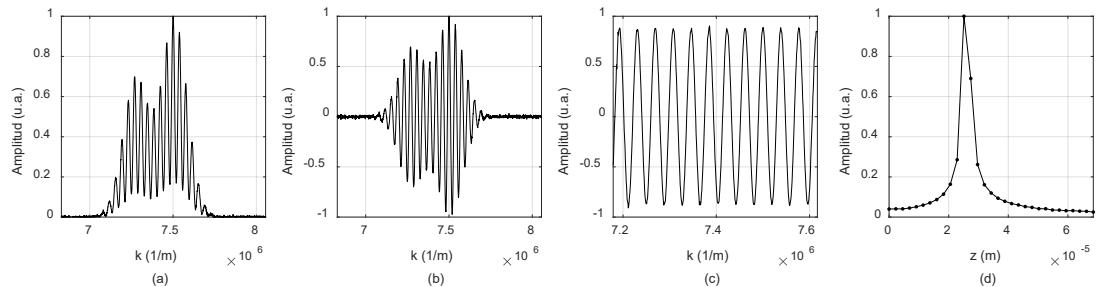


Fig. 177 – Preprocesado convencional, primero se detecta la señal de LCI, luego se filtra las frecuencias bajas, se identifica la envolvente y divide finalmente realizar la transformada rápida de Fourier de la señal oscilante.

Al realizar la fft la resolución espectral queda determinada por el número de muestras de la trama de la señal original, como en el caso de la Fig. 177, esto puede generar una limitación en la detección del máximo del pico y por lo tanto un error en la determinación de la posición. Este efecto se puede corregir utilizando técnicas como zero padding. En particular la técnica de zero padding implica un agregado de datos a la trama original por lo tanto se trabaja con vectores más grandes incrementando la carga computacional. Una alternativa es utilizar la transformada chirp de Fourier, en este caso se permite seleccionar el rango de frecuencias donde se desea obtener la transformación y se indica la cantidad de puntos deseados entre ese par de frecuencias, esto mejora la resolución, pero imposibilita realizar la anti transformada.

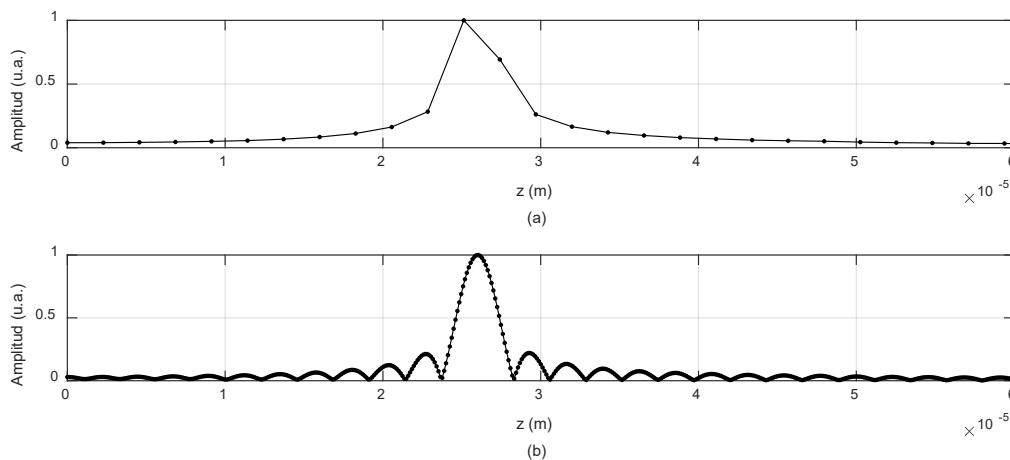


Fig. 178 – Comparación de la transformada de Fourier de una señal de LCI para menor (arriba) y mayor (abajo) número de muestras. Tomando mayor número de muestras se puede identificar la posición del máximo con mejor precisión

Al aplicar la transformada chirp se evidencian adicionalmente los efectos de la cantidad de muestras del frame, como se puede observar en la Fig. 178.

Estimación de frecuencia

Como se vio anteriormente, la determinación de la posición del pico de la transformada de Fourier indica la información deseada, claramente esa información también se puede obtener al medir los periodos de la señal en el espacio k, por lo cual el problema central que se desea resolver es la estimación de frecuencias.

La estimación de la frecuencia de una onda sinusoidal ruidosa con precisión ha sido uno de los principales problemas en el campo del procesamiento de señales y las comunicaciones, debido a sus amplias aplicaciones que incluyen sistemas de energía (Routray et al., 2002), comunicaciones (Classen & Meyr, 1994), radar (Orović et al., 2010)(Bamler, 1991)(Liu et al., 2012) , vibraciones mecánicas, acústica, ultra sonido, medicina, entre los más destacados.

Se han propuesto muchas técnicas teóricas para resolver este problema; los ejemplos incluyen la transformada discreta de Fourier (Belega & Dallet, 2008), los métodos de mínimos cuadrados (Yardibi et al., 2010) y los lazos de fase bloqueada (Martínez-Montejano et al., 2014). Todos los métodos propuestos se centran en la velocidad y precisión de la estimación.

El uso de redes neuronales ya se ha introducido en el campo del procesamiento de señales (Deng, 2008) (Tang & Eliasmith, 2010) (LeCun et al., 2007) y en temáticas asociadas a frecuencias por ejemplo para encontrar la estimación de máscara de relación ideal para filtrar el ruido de un espectrograma, monitoreo de frecuencia en tiempo real (Lai et al., 1999), detección de canal en sistemas de multiplicación por división de frecuencia ortogonal (Gao et al., 2018), y en la obtención de retroalimentación de información de estado de canal de sistemas masivos de múltiples entradas y múltiples salidas (T. Wang et al., 2019). Pero hasta la actualidad, son pocos y recientes los trabajos que se centraron en estimar directamente la frecuencia de señales mediante redes neuronales. En (Sajedian & Rho, 2019) se encuentra que mediante una red neuronal sencilla densamente conectada se pueden estimar frecuencias de señales senoidales con presencia de ruido. Esta técnica se analiza en mayor profundidad en (Almayyali & Hussain, 2021) y se hace una comparación con las trabajos existentes tales como (Xu & Shimodaira, 2019) en el cual se estima la frecuencia fundamental de una

señal y en (Chen et al., 2019) donde se estiman señales linealmente moduladas en frecuencias mediante redes convolucionales.

Otra ventaja es que las técnicas que usan la transformada de Fourier discreta (DFT) requieren el uso de una aritmética de valor complejo, mientras que los sistemas de redes neuronales usan solo una aritmética de valor real. La implementación de una DFT para grandes cantidades de datos es una tarea muy complicada para los diseños de hardware y software a pesar de la eficiencia significativa proporcionada a través de las últimas versiones del algoritmo de transformada rápida de Fourier, que reduce el número de órdenes de multiplicaciones (complejas) de N^2 a $N\log(N)$ donde N es el número de muestras.

Es claro que se puede entrenar a una red neuronal para realizar la transformada de Fourier, como se destaca en (Rosemarie, 2008), pero no sería el método más eficiente, ya que como se mencionó los algoritmos existentes tienen gran eficiencia y flexibilidad. La idea es entonces hacer que la red neuronal aprenda a estimar la frecuencia, generando a partir de su aprendizaje su propio método de estimación.

Consideraciones generales

Se toma como elemento de análisis la señal descripta en la ecuación (11.16) que describe la intensidad de la señal de interferencia detectada por el espectrómetro para la condición de perfilometría.

Sin lugar a duda el éxito de la implementación de una Red Neuronal Artificial (de ahora en más la abreviaremos como RN sin la necesidad de explicitar que son artificiales) depende de la cantidad de datos que tengamos para entrenar. Los datos para entrenar la RN se pueden obtener, simplificando un poco, por dos caminos, el primero por la medición directa, en particular este es un proceso bastante costoso en términos de tiempo, pero con la ventaja de que los datos son reales. Un segundo camino es la simulación, que presenta la ventaja de ser un método más rápido para la obtención masiva de datos de entrenamiento, con una desventaja que es la utilización de datos no reales. Se debe tener en cuenta que las simulaciones permiten acelerar los procesos de diseño de la RN, y ser modificadas las veces necesarias para asemejarse a la señal real.

En este apartado se trabajó sobre señales simuladas, por este motivo es que se debe tener un buen entendimiento de los fenómenos físicos en estudio para garantizar que la programación de las señales genere simulaciones realistas. Se simularon señales con espesores z_s comprendidos entre 100 μm a 1000 μm y con z_s comprendidos entre 30 μm a 500 μm dando exactamente el doble para las DCO, considerando la forma espectral de la fuente de luz gaussiana centrada en 850 nm y con un ancho de 50 nm. Calculando mediante la ecuación (11.13) se obtiene una resolución de 8 μm .

Un tema que debe estar en consideración son las capacidades del hardware disponible. Esto juega un rol crucial en los tiempos que lleva generar las simulaciones de las señales, los entrenamientos y ejecuciones de las RN. Las operaciones presentes en este capítulo fueron realizadas en una computadora de 2 núcleos y procesador de 2.20 GHz Intel(R) Core (TM) con 8 GB de RAM. Los diseños y entrenamiento de las RN que se proponen a lo largo del capítulo fueron pensados para que puedan ser realizados en una computadora promedio personal y que el proceso no demande más de 24hs.

El diseño, entrenamiento y manipulación de las RNs que se presentan fueron realizados en Matlab. Fácilmente pueden ser extendidos a Python utilizando los paquetes de TensorFlow y Keras.

Para medir el desempeño de las RN implementadas se usó el error cuadrático medio RMSE del subconjunto de testeo entre los valores reales Y_{test} y los predecidos Y_{pred} .

$$RMSE = \frac{1}{T_{test}} \sqrt{\sum_{i=1}^{T_{test}} (Y_{test_i} - Y_{pred_i})^2} \quad (11.17)$$

Vale la pena aclarar que al repetir el entrenamiento de una misma red con exactamente los mismos parámetros el valor de RMSE puede tener una leve variación debido a que los valores elegidos de testeo Y_{test} varían de manera aleatoria dentro de todo el conjunto total de datos. Otro parámetro que se utilizó para caracterizar la efectividad de las redes fue el error con respecto al peor valor predicho RSE_{max} sobre todo el conjunto Y (entrenamiento, validación y testeо).

$$RSE_{max} = \max \left\{ \sqrt{(Y - Y_{pred})^2} \right\} \quad (11.18)$$

Para poder comparar los tiempos de ejecución de las distintas RN con la forma convencional de procesar la señal se calcularon los tiempos que demora cada etapa, ver Tabla XVII.

Tabla XVII – Tiempo que demora cada etapa de preprocesada para la estimación de frecuencia para la técnica de LCI. Se marcan los órdenes de magnitud ya que *la precisión de los tiempos depende fuertemente del procesador.

Etapa de procesado	Tiempo [ms]*	Orden de magnitud
Eliminación de continua	1.10	ms
Determinación de envolvente	2.31	ms
Normalización	0.06	decenas de μ s
Recorte	1.52	ms
FFT	0.08	decenas de μ s
Detección de pico	0.01	decenas de μ s

Estimación paramétrica de frecuencias con redes neuronales

La estimación paramétrica se caracteriza por utilizar como elementos de entrada a la RN un conjunto de parámetros característicos de la señal, de manera tal que la cantidad de entradas a la red es pequeña, la conformación de la red neuronal se dice que es sencilla en cuanto al diseño de la cantidad de neuronas por capa, cantidad de capas y tipos de las neuronas. Esto claramente no implica que necesariamente sea más simple en cuanto al procesado de la información.

Esta propuesta consiste en obtener parámetros relevantes de nuestra señal de interés mediante un preprocesamiento de la señal. Cuanto más característicos y/o peculiares sean los parámetros elegidos de la señal mayor va a ser la efectividad en el entrenamiento de la red consiguiendo una óptima especificidad y selectividad en las predicciones. Es por este motivo que vale la pena tomarse un tiempo para pensar:

- ¿Cuáles son los parámetros más importantes?
- ¿Qué parámetros constituyen la huella digital de mi señal?
- ¿Cuáles son las características más significativas y de mayor variabilidad?

Las señales temporales tienen características típicas bien identificables como:

- La intensidad máxima.
- La distancia entre picos.
- El ancho de cada pico.
- El valor de la señal pico a pico.
- Valor de fase.
- Energía y/o potencia.

Se utiliza para esta descripción y a modo de ejemplo ilustrativo del análisis de señales que se desea analizar con una RN, una señal de oscilante típica de interferometría de baja coherencia, Ecuación (11.16). Se utilizaron 1000 simulaciones de señales típicas de LCI para valores de espesores z_s equiespaciados entre 100 a 1000 μm y se extrajeron cuatro parámetros de interés:

- La intensidad máxima global (I_{\max}).
- El valor pico a pico (VPP) en las cercanías del máximo global.
- El promedio de separación entre dos máximos locales consecutivos (PSM).
- El promedio del ancho de cada uno de los máximos (PAM).

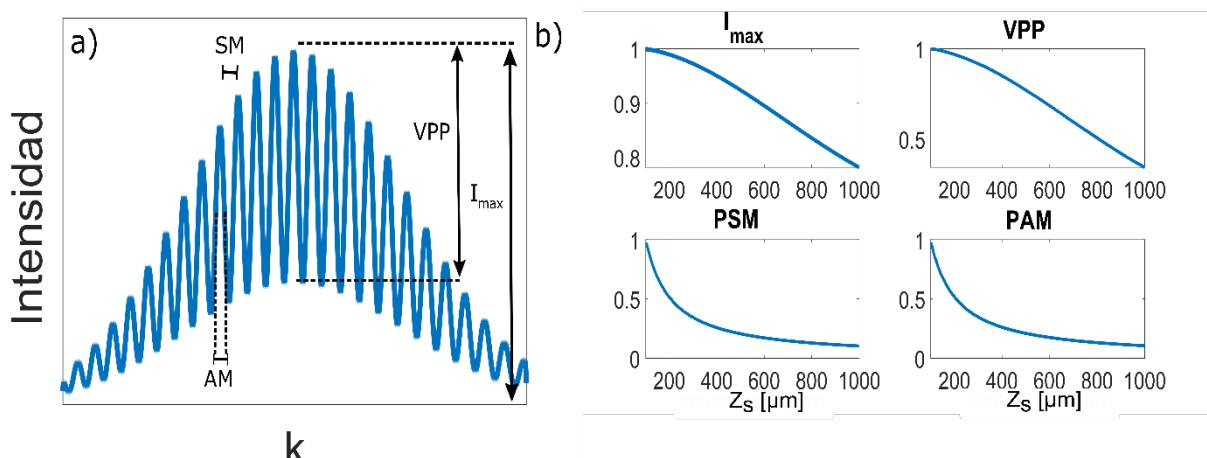


Fig. 179 – a) Esquema ilustrativo con los parámetros elegidos para caracterizar una señal típica de LCI. b) Valores de los parámetros para las 1000 simulaciones realizadas variando la distancia z_s .

Se diseñó una RN sencilla (o también llamada “shallow”) del tipo feedforward conformada por la capa de entrada que cuenta con tantas neuronas como parámetros de entrada (Input) deseados, seguido de una capa interna de 8 neuronas y una capa de salida de una única neurona para obtener una respuesta del tipo regresiva, Fig. 180.

Esto permite no solo usar a la RN para clasificar conjuntos de entradas (Inputs) dentro de ciertas categorías definidas, si no también obtener valores continuos como datos de salida (output) para cada conjunto de inputs. Notar que este trabajo lo realiza la función de activación de la neurona de la capa de salida. En general no se suele aclarar la conformación de la capa de entrada porque queda definida por el número de valores que conforman cada Input ni de la capa de salida ya que queda definida por el tipo y cantidad de salidas deseadas.

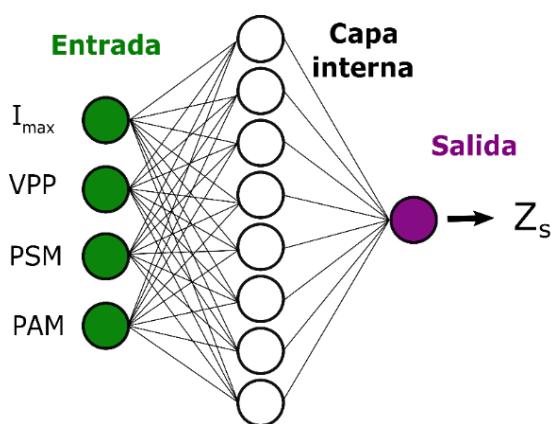


Fig. 180 – Estructura de la Red Neuronal de 8 neuronas densamente conectadas elegida para estimar frecuencia de señales parametrizadas. Se toman 4 valores de entrada y uno de salida.

Con un preprocesamiento de la señal se obtienen los 4 parámetros seleccionados para realizar el entrenamiento de la RN. Esta RN luego de ser entrenada puede obtener una estimación de la frecuencia de oscilación con una alta precisión.

Se utilizó el algoritmo de optimización para el entrenamiento de Levenberg-Marquardt. Para el diseño de la RN de 8 neuronas se usó la función *fitnet* que devuelve una función que se ajusta a la red neuronal con el tamaño de capa oculta seleccionado y se entrenó con la función *train(net, X, Y)* donde *net* es la red diseñada, *X* es una matriz con los datos de entrada de 7x1000 e *Y* es un vector de 1x1000 con los valores de frecuencia de la simulación requeridos para entrenar. El 72% de los datos se usó para el entrenamiento, el 18% para validación y el 10% para testeo. Se predijeron con la red entrenada los datos de testeo y se obtuvo un RMSE = 0.10 μm .

Un análisis para este caso puede resultar poco redundante ya que parte del preprocesado es calcular la distancia entre máximos y la inversa de ese valor es la frecuencia en cuestión $z_s = 1/PSM$. En principio no sería necesario predecirlo con una RN. El objetivo era mostrar un ejemplo ilustrativo sencillo para comprender mejor la metodología de trabajo.

Por otro lado, puede resultar interesante usar RN en un caso similar, si se desea ponderar el valor de la frecuencia con algún (o algunos) parámetro asociado como el VPP o si se desea calibrar el detector a partir de mediciones conocidas. En general en el caso de LCI, y en otras aplicaciones también, la señal posee agregados y artefactos donde el uso de RN cobra una mayor utilidad y es uno de los temas que se trata en este capítulo.

En una señal real de LCI, como se comentó en la sección de Interferencia de Baja Coherencia, tanto los dispositivos de detección como la muestra en estudio pueden generar efectos no deseados o que modifican a la señal.

- a) La dispersión angular distinta para cada longitud de onda en la red de difracción sumada a la llegada de un frente de onda esférico (o gaussiano) a un detector con geometría plana, genera un muestreo no uniforme de la señal de LCI.
- b) Si el material en estudio tiene un índice de refracción $n(\lambda)$ que varía con la longitud de onda, genera un efecto dispersivo a la señal de LCI.

El efecto visible que estos efectos individuales generan en la señal de LCI es similar, en ambos casos, una deformación del pico en la transformada de Fourier, que genera errores en la detección de la frecuencia.

Para probar el método paramétrico se utilizó la señal de LCI con presencia de ruido. Teniendo en consideración que los métodos para corregir los efectos del muestro no uniforme no son costosos y se encuentran bien establecidos, se utiliza entonces una señal de LCI considerando solo el efecto de índice de refracción $n(\lambda)$.

Se generó un conjunto de señales de LCI con el índice de refracción $n(\lambda)$ de un vidrio BK7 y se amplificó por 5 para evidenciar el efecto dispersivo, se puede observar en la Fig. 181.

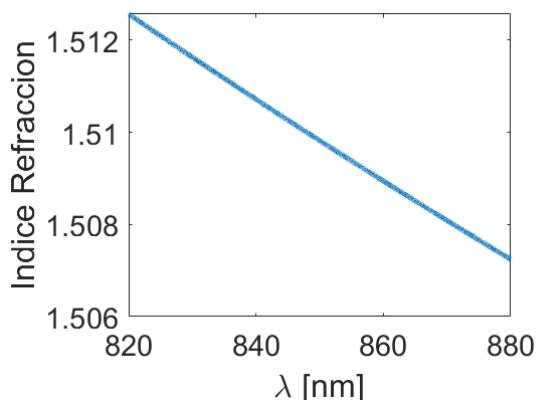


Fig. 181 – Índice de refracción en función de la longitud de onda utilizado en la simulación

Como la DCO aumenta al tomar valores de n mayores para iguales D , se disminuyó el efecto del Fall-off para que las señales simuladas tengan un VPP apreciable.

Al utilizar una señal más compleja, la obtención de los parámetros se vuelve más sofisticada al igual que el preprocesado de la señal de LCI para la obtención de los parámetros con precisión (filtrar espectralmente, calcular envolventes, agregar restricción para calcular mejor la posición de los máximos de las señales, etc.).

A diferencia del ejemplo anterior la señal de LCI presenta el efecto de dispersión por lo cual se agregaron entradas adicionales:

- La desviación estándar del ancho promedio de los picos.
- La desviación estándar de la separación entre máximos.

Como se observa en la Fig. 182 los valores I_{\max} y V_{PP} tiene mayor variación debido a la incorporación de ruido.

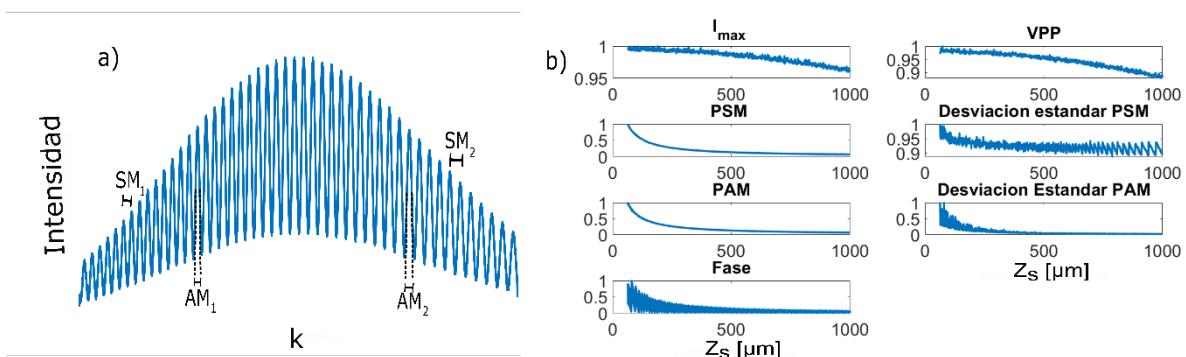


Fig. 182 – a) Esquema de una señal de LCI experimental con chirpeo, notar que $SM_1 < SM_2$ y $AM_1 < AM_2$ para una distancia z_s . b) Valores de los 7 parámetros para las 1000 simulaciones realizadas variando la Distancia Z_s .

Se utilizó la misma configuración de RN que en la Fig. 180 y el mismo proceso de simulaciones, pero tomando ahora 7 valores de input. La función *train* tiene la ventaja que aadecua

automáticamente la cantidad de entradas que requiere la RN a partir de las dimensiones de X y T prácticamente sin la necesidad de alterar el código.

Al entrenar la RN con las 1000 simulaciones se obtuvo un RMSE = 1.1 μm probando que sigue siendo una buena manera de estimar la frecuencia. Los tiempos de las simulaciones y preprocesado dependen fuertemente del tamaño del vector k que queda definido por Δk y k_m . Considerando un tamaño de k de 1x10.000. Los tiempos de procesamiento son: 4.65 minutos para las simulaciones de la señal, 30 segundos para el preprocesado de todas las señales ($\frac{30 \text{ s}}{1000} = 30 \text{ ms}$ cada una), 16 segundos para el entrenamiento y 9 ms para la ejecución de un conjunto de entrada teniendo la red ya entrenada.

Es evidente que a medida que las señales se tornan más complejas y/o se intenta que la RN permita resolver un conjunto más grande de problemas el preprocesado necesario para la obtención de los parámetros se vuelve cada vez más importante. Adicionalmente, se debe destacar que cada material tiene un índice de refracción particular, por lo cual se deberá tener un conjunto de señales de entrenamiento y preprocesado adecuado para cada material. Se pudo probar que con una RN simple se puede obtener una muy buena estimación en frecuencia por debajo de la resolución esperada típica de LCI.

Estimación de frecuencias a partir de la señal completa

Las preguntas que se desean contestar son:

- ¿Se puede evitar la necesidad de seleccionar parámetros?
- ¿Se puede independizar del preprocesado
- ¿Cómo se puede lograr?
- ¿Qué tipo de RN se deben utilizar?

En esta sección se irán implementando diversas configuraciones de RNs para ir evitando en distintos grados las instancias de preprocesado sobre la señal original de LCI.

La primera diferencia con respecto al caso anterior es que ahora los datos de entrada a la RN serán los datos muestreados de la señal de LCI, de esta manera se evita la selección de parámetros que tienen que preestablecerse con antelación. En el caso de LCI la señal es detectada utilizando un espectrómetro como se describe en la sección anterior, el detector CCD de los espectrómetros pueden variar desde 512 a más de 12k pixeles, según los diferentes modelos presentes en el mercado.

Las características del espectrómetro determinan entre otros parámetros la frecuencia de muestreo y la cantidad de muestras (ventana espacial).

Se destaca que de acuerdo con el teorema de Nyquist-Shannon, la frecuencia máxima de la señal está determinada por:

$$f_m \geq 2f_{s_{max}} \quad (11.19)$$

Donde f_m es la frecuencia de muestreo y $f_{s_{max}}$ es la frecuencia máxima de señal para poder reconstruir sin distorsión. Este marco teórico es necesario en el proceso de análisis que se desea realizar, ya que al tener fija la f_m si las señales se encuentran a frecuencias superiores de $f_{s_{max}}$ aparecerían problemas de aliasing que ocasionan interpretaciones erróneas.

Otro aspecto para tener en cuenta es la resolución, que de acuerdo con la teoría del análisis de señales se puede definir como “al mínimo salto en frecuencia (Δf) que se puede detectar correctamente sin alteraciones”, matemáticamente se describe como:

$$\Delta f = \frac{f_m}{N} \quad (11.20)$$

Donde N es la cantidad de muestras adquiridas en la ventana de adquisición. Un ejemplo que puede resultar esclarecedor es: si tenemos una señal de audio típicamente muestreadas a 48 kHz y durante la ventana temporal se adquieren 128 muestras, la fft tendrá una frecuencia de resolución de 375Hz (48000/128). Por tanto, no se deberán considerar ninguna señal por debajo 375Hz.

Por otro lado, la señal de LCI tiene una envolvente, en general de tipo gaussiana, por lo cual la frecuencia mínima detectable estará condicionada al ancho de dicha envolvente

En una primera aproximación y teniendo en cuenta a la Fig. 176, se considera que la señal de entrada a la RN se encuentra preprocesada como indica la Fig. 183.

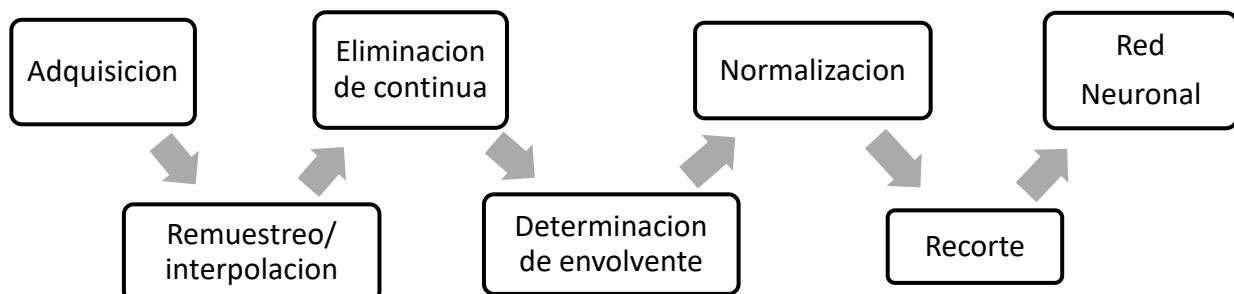


Fig. 183 – Esquema inicial de las etapas de preprocesado de una señal LCI antes de ser introducida a la red neuronal.

Por lo tanto, la señal de LCI a la entrada de la red neuronal queda definida como:

$$i(k) = \cos(kDCO) = \cos(k2z_s) \quad (11.21)$$

Donde z_s es el valor que se desea estimar, como n=1 la DCO es el doble del valor de z_s . Una situación muy similar se encuentra estudiada en mayor profundidad en (Sajedian & Rho, 2019) pero considerando señales temporales. Los autores luego de probar diferentes diseños de redes proponen uno asombrosamente sencillo capaz de estimar frecuencias, dentro de ciertos rangos. Una de las características más interesantes que se propone en (Sajedian & Rho, 2019) es analizar un rango de frecuencias que estén comprendidas dentro de un orden de magnitud, toma como ejemplo señales

entre 1 kHz a 10 kHz tomando 2000 muestras con una $f_m = 1$ MHz. Esta selección permite, por un lado, visualizar por lo menos dos oscilaciones completas para la frecuencia más lenta y, por otro lado, muestrear 100 veces la oscilación para la frecuencia más rápida. También amplia el rango de señales para la cual RN ya entrenada puede utilizarse ya que en caso de tener señales en otros rangos de frecuencias hay que cambiar la escala y la relación de la figura de entrada a la red sea la misma. Sin embargo, también es posible entrenar la RN propuesta para estimar rangos de frecuencia mayores. En el trabajo de (Sajedian & Rho, 2019) se utilizó una RN con 2000 elementos en la capa de entrada, 3 capas densamente conectadas o también llamadas “fully-connected” de 2, 2 y 3 neuronas cada una (también denominadas “Hidden layers”) y una neurona como capa de salida para obtener un único resultado regresivo como valor de frecuencia.

A lo largo de esta sección se trabajó con simulaciones de señales con z_s comprendidas entre 30 a 500 μm (es decir DCO de 60 a 1000 μm) tomando 30.000 o 60.000 o 100.000 valores equiespaciados en ese rango. Se utilizó una ventana espacial de 2000 muestras y una $f_m = \frac{1}{k_m} = 9.3$ mm. Esto permite que para la frecuencia mínima (DCO = 60 μm) se observen dos ciclos completos de la señal y que la señal de mayor frecuencia tenga 60 muestras por ciclo ya que $f_{max} \ll f_m$. Esto permite generar datos de entrada significativos para el aprendizaje de la RN.

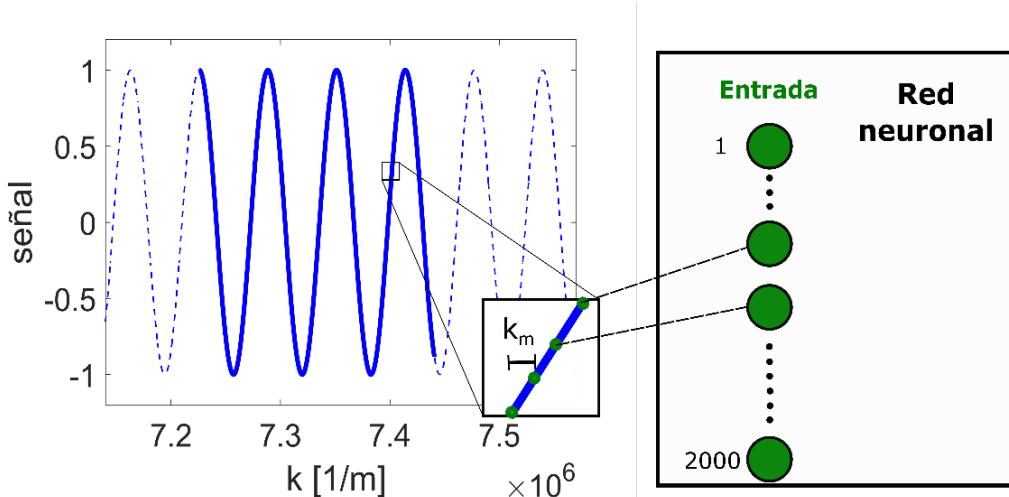


Fig. 184 – Muestreo de la señal a procesar para las diferentes redes neuronales. Los puntos 2000 se encuentran separados una distancia k_m y son los valores de entrada a la red neuronal. El ejemplo en este caso es una señal senoidal con un z_s de 50 μm .

Para el entrenamiento de todas las RN presentes en esta sección, a cada conjunto de señales se las separa de modo aleatorio de la siguiente manera: el 72% para entrenamiento, el 18% para validación y el 10% restante para testeо. Se utilizó en su mayoría el algoritmo de optimización Adam con un coeficiente de aprendizaje de 0.01 y el gradiente estocástico descendente con momento (sgdm) se utilizó en dos casos, que se lo detalla explícitamente.

De acuerdo a los buenos resultados obtenidos en (Sajedian & Rho, 2019) se decidió diseñar una RN igual, ver Fig. 183, y probar su funcionamiento para la ecuación (11.21). Se utilizó un conjunto de 100.000 señales, se entrenó para 60 iteraciones (epochs) con la función de entrenamiento Adam con coeficiente de entrenamiento inicial de 0.01. Como se puede observar en la Fig. 187, se obtuvo un $RMSE = 2.3$ μm y un $RSE_{max} = 14.8$ μm , el tiempo del entrenamiento demoro 7 minutos y la RN una vez ya entrenada tiene un tiempo de ejecución 2.0 ms para estimar el resultado. La simulación de las 100.000 señales demoro 40 minutos aproximadamente. Este es un muy buen

resultado ya que se encuentra para la mayoría de los valores de z_s salvo los que se encuentran en los extremos por debajo de la resolución típica de la técnica de LCI. Este comportamiento suele suceder en los extremos debido a que no existen valores con los que se puedan seguir entrenando a la RN. Se podría solucionar agrandando el rango de valores de z_s para el entrenamiento y luego recortando los extremos.

Un detalle que siempre ayuda al entrenamiento de las redes es normalizar los valores para que los outputs estén entre 0 y 1. Simplemente es dividir el valor de los outputs por su máximo y también recordar multiplicar por este mismo a los valores predichos.

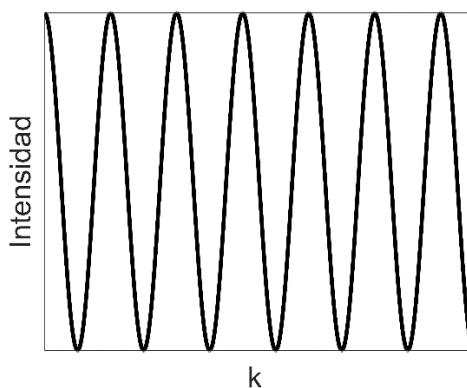


Fig. 185 – Intensidad de una señal cosenoide en función de k. La frecuencia de la oscilación está dada por la distancia z_s

Entre las funciones utilizadas para el diseño y entrenamiento de las RNs se destacan: *imageInputLayer* que permite manejar de manera sencilla el gran número de inputs. Si bien sirve para manipular imágenes también se puede adaptar para señales unidimensionales, *layers* para el diseño de las sucesivas capas que componen a la RN y *trainNetwork* para el entrenamiento de redes neuronales de aprendizaje profundo.

El código utilizado para el armado de las capas queda entonces de la siguiente manera:

```
layers = [ ...
    imageInputLayer([2000 1 1])
    fullyConnectedLayer(2)
    fullyConnectedLayer(2)
    fullyConnectedLayer(3)
    fullyConnectedLayer(1)
    regressionLayer];
```

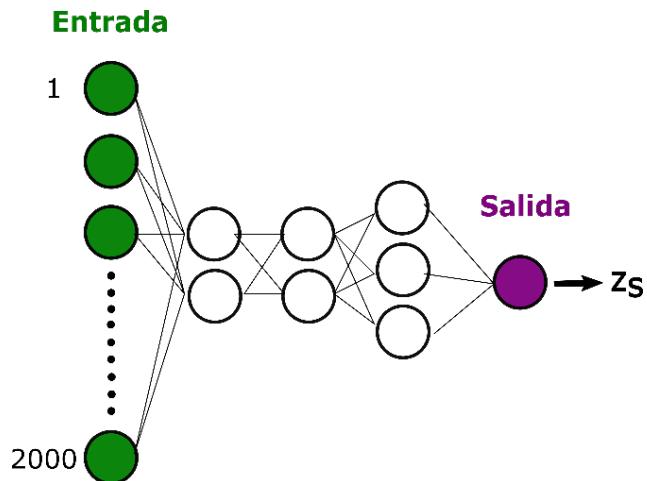


Fig. 186 – Esquema de la una red neuronal de tres capas densamente conectadas de 2,2 y 3 neuronas con salida regresiva.

Notar que no sirve usar la RN ya entrenada para estimar la frecuencia de otro tipo de señal. Por ejemplo, si se utiliza esta misma RN para predecir los valores de testeo, pero multiplicados por una amplitud de $A=2$ o $A=5$ se obtienen RMSE mayores a $100 \mu\text{m}$. Es por este motivo que cada conjunto de señales requiere su entrenamiento apropiado. Evidenciando además que la normalización juega un rol importante.

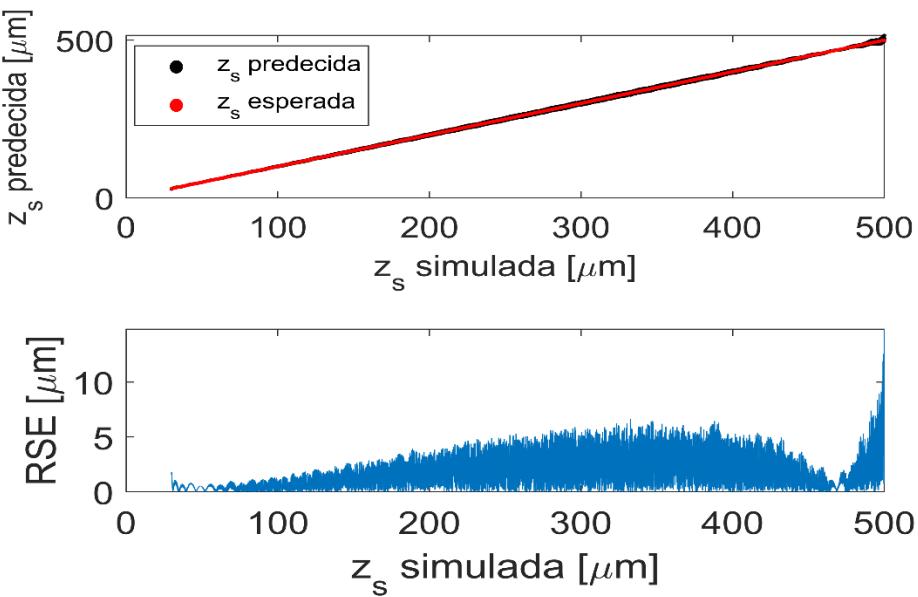


Fig. 187 – predicción para 100.000 señales coskOPD para red 223 para 60 epochs. RMSE=2.3 μm y $RSE_{max}=14.8 \mu\text{m}$

Un problema que se logró identificar es que esta RN no funciona adecuadamente para el caso en que la señal de LCI presenta fase, en la Fig. 188 se puede observar el grafico de esta señal y en la ecuación (11.22) su expresión matemática.

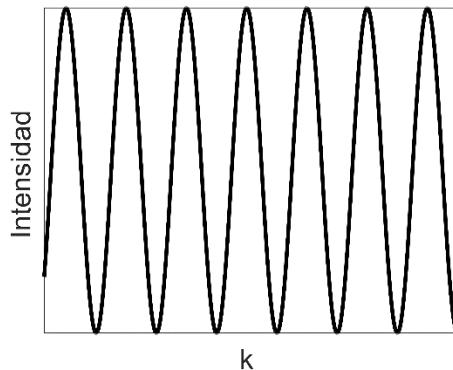


Fig. 188 – Esquema de una señal cosenoide con la presencia de un desplazamiento debido a la fase extra.

$$i(k) = \cos(kDCO + \varphi) \quad (11.22)$$

Si las señales están desfasadas por φ tanto aleatoriamente como con cierta relación constante entre sí la RN no puede estimar la frecuencia con buena precisión con este tipo de red. La aparición de fase φ en las señales de LCI se pueden deber al sistema de medición o a la muestra en estudio. En la Fig. 189 se puede apreciar el valor estimado por la RN y el valor teórico, claramente la estimación de este modelo de RN es de mala calidad.

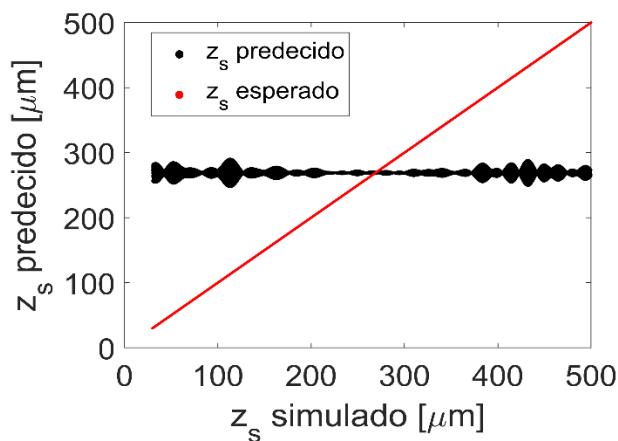


Fig. 189 – Se puede observar la mala predicción (negro) de la FCN 2-2-3 para señales que presentan fase distinta

El problema de la fase se puede solucionar con un preprocesamiento de la señal. En cuyo caso se utiliza una ventana deslizante con 2000 puntos que se ajusta su posición para que la señal de LCI siempre inicie desde fase cero.

Lo que se desea evidenciar con esta situación es que si bien la RN fully-connected o densamente conectada de capas de 2 neuronas, 2 neuronas y 3 neuronas (FCN 223) tiene gran capacidad de aprendizaje nunca va a poder aprender en situación donde las señales tengan un desfasaje entre ellas. Es por esto por lo que hay que recurrir a nuevos diseños de RN si se desean solucionar esta problemática.

Se pueden utilizar configuración de RN más elaboradas agregando más capas y más neuronas por capa del tipo fully-Connected (Almayyali & Hussain, 2021), agregando capas de activación. Si bien esto incrementa el tiempo requerido para entrenamiento permite mejorar la precisión de la predicción de la red.

Por lo tanto, se diseñó una RN de 7 capas con numero decreciente de neuronas por capa 1000, 500, 250, 125, 50, 25 y 10. Como se puede observar en la Fig. 190.

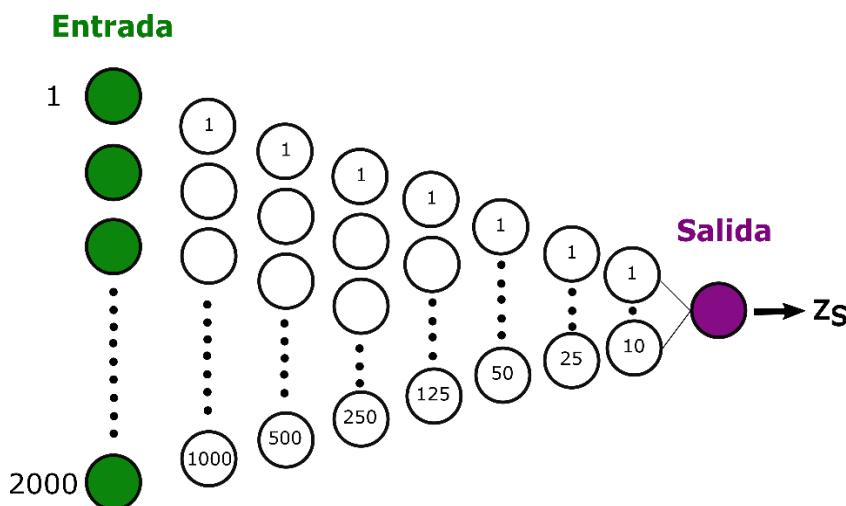


Fig. 190 – Esquema de la red densamente conectada FCN 100-500-250-125-50-25-10

Entrenando a este tipo de red con mayor número de neuronas, capas y funciones de activación del tipo tangente hiperbólica, como la que se describe a continuación:

```
layers = [
    imageInputLayer([2000 1 1])
    fullyConnectedLayer(1000)
    tanhLayer
    fullyConnectedLayer(500)
    tanhLayer
    fullyConnectedLayer(250)
    tanhLayer]
```

```

fullyConnectedLayer(125)
tanhLayer
fullyConnectedLayer(50)
tanhLayer
fullyConnectedLayer(25)
tanhLayer
fullyConnectedLayer(10)
tanhLayer
fullyConnectedLayer(1)
tanhLayer
regressionLayer];

```

Donde después de cada etapa `fullyConnected` se utiliza una `TanhLayer` que actúa como una capa de activación de tangente hiperbólica. Para el entrenamiento de esta RN se usó el conjunto de 100.000 señales, 90 epochs, con función de entrenamiento se utilizó el sgdm, una tasa de aprendizaje de 0.1 y una frecuencia de validación de 100. Se obtuvo un RMSE=2.2 μm y $\text{RSE}_{\text{MAX}} = 16.5 \mu\text{m}$. El entrenamiento demoró 27 minutos y el tiempo de 4.8 ms En la Fig. 191, se pueden observar los resultados predichos y los valores esperados, claramente hay un gran nivel de coincidencia, además se puede observar que las variaciones más grandes se encuentran en los extremos, es decir para los valores de frecuencia más pequeños y valores de frecuencia más grandes. También se observa la variación del RSE para cada valor proporcionado por la red.

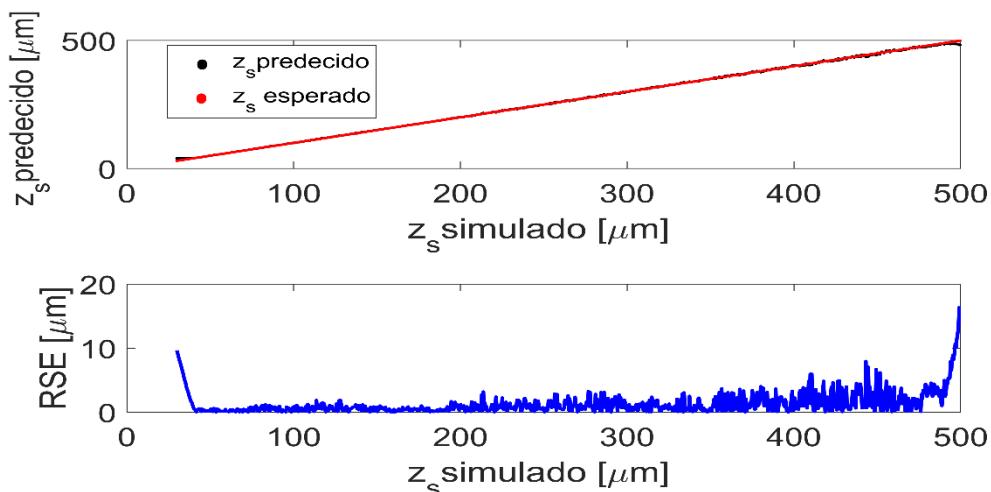


Fig. 191 – predicción y error para 100.000 señales $\text{Cos}(kDCO + \varphi)$ para una red densamente conectada de 1000-500-250-125-50-25-10 neuronas.

Los resultados anteriores son alentadores pensando en el objetivo final buscado que es el de eliminar todo el sistema de preprocessamiento y que sea la RN la única encargada de realizar el análisis sobre la señal de LCI.

En este punto, la RN ha demostrado poder determinar las frecuencias de las señales de LCI con fase, en el sistema de preprocesado ya no resulta necesaria la etapa de recorte quedando el esquema de la Fig. 192:

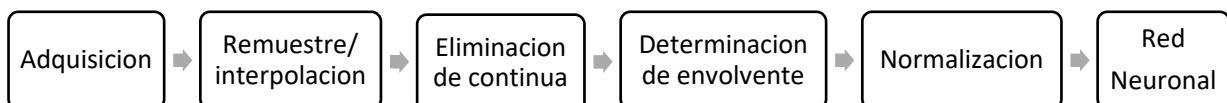


Fig. 192 – Esquema de preprocesado de la señal. Modificando la calidad de la red neuronal se puede evitar la etapa de recorte.

Una etapa interesante de eliminar es la etapa de eliminación de envolvente, la señal antes de ingresar en esta etapa está descripta por la Ecuación (11.23) y se puede observar su gráfico en la Fig. 193.

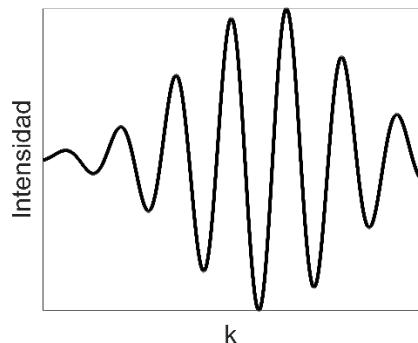


Fig. 193 – Esquema de una señal de LCI sin la componente de baja frecuencia. La cual posee una oscilación cosenoidal con una envolvente.

$$i(k) = s(k) \cos(kDCO + \varphi) \quad (11.23)$$

Por lo cual se entrenó la RN anterior con este tipo de señales para determinar la efectividad de su utilización. Se plantean condiciones similares a las que se utilizaron con anterioridad para el entrenamiento de la RN, se toma un conjunto 30.000 de señales de entrenamiento, los resultados se pueden ver en la Fig. 194 donde se puede observar que las frecuencias esperadas y las frecuencias estimadas difieren notoriamente, se obtuvo un RMSE=13.1 μm y RSE_{MAX} mayor a 100 μm este modelo de RN, no es apropiada para señales con una componente que le varíe la amplitud para distintos valores de k , como lo indica la ecuación (11.23), si se toman más cantidades de señales se sigue observando un comportamiento similar.

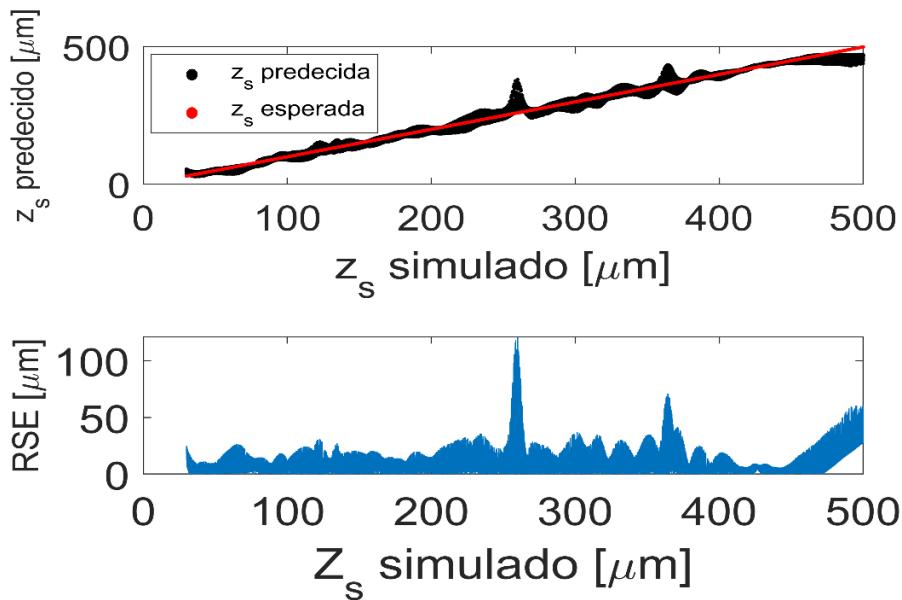


Fig. 194 - Predicciones para señales del tipo $s(k) \cos(kDCO + \varphi)$ para la red densamente conectada de 1000-500-250-125-50-25-10 neuronas.

Seguir agregando capas y neuronas densamente conectadas no consigue solucionar este problema. Si se desea solucionar esta problemática resulta interesante empezar a considerar RNs que tengan capacidades de aprendizaje distintas. Que su diseño y manera de aprender les permita reconocer patrones de mayor complejidad. Es necesario ir más profundo o como se comenta en una de las publicaciones más citadas sobre Redes Neuronales “We need to go deeper” (Szegedy et al., 2015). Si bien esa frase se usa en un contexto un poco diferente a este, aplica para la situación en que estamos. La propuesta para ir más profundo es usar Redes Convolucionales.

No está de más remarcar, que las RNs propuestas a lo largo de todo este capítulo tienen la capacidad de estimar frecuencias de señales oscilantes sea cual sea su naturaleza. Se estuvo analizando particularmente señales de LCI, pero con las mismas RNs se pueden analizar cualquier otra problemática donde haya una única oscilación y se desea estimar su frecuencia. Por ejemplo, podrían servir para estudiar el movimiento de un péndulo pasando por la tensión eléctrica de entrada a las casas hasta llegar una comunicación satelital.

Ganando profundidad para estimar frecuencias de señales completas

Las redes neuronales convolucionales son un tipo de RN diseñado originalmente para procesar datos de imágenes, pero las mismas propiedades que hacen propicias para problemas de visión las hacen muy relevantes para procesar señales. Se puede extrapolar entendiendo al tiempo como una dimensión espacial, esta línea ha nacido recientemente y es denominada redes convolucionales en 1D, trabajos actuales han demostrado que para ciertas aplicaciones se puede obtener resultados similares que usando convoluciones en 2D (Kiranyaz et al., 2021) (Kiranyaz et al., 2019). La gran ventaja es que la configuración simple y compacta, debido a sus operaciones unidimensionales permite implementaciones en tiempo real y con menores requerimientos del hardware.

Las capas de convolución en una dimensión obtienen nuevas secuencias a través de filtros que interpretan ciertas características de las secuencias originales que permiten reconocer patrones locales en la misma. Son altamente utilizadas en aplicaciones como clasificación de datos biomédicos y diagnóstico temprano, detección e identificación de anomalías se señales electrónicas, detección de fallas en motores eléctricos, etc.

En este sentido la capa convolucional es, tal vez, la capa más significativa de este tipo de RNs. La diferencia fundamental entre una capa densamente conectada y una capa convolucional es que las capas densas aprenden patrones globales de sus entradas, mientras que las convolucionales aprenden filtros (o Kernels) que modifican la señal original, generando descriptores o mapas de características.

Esta arquitectura le permite a la red concentrarse en características más simples en las primeras capas y agruparlas luego en características más complejas en las capas siguientes. Las capas convolucionales se componen de estructuras, filtros y mapas de características. Los filtros son esencialmente los distintos grupos de pesos sinápticos de las neuronas de la capa. Durante el entrenamiento, la red neuronal convolucional encuentra los filtros más útiles para su tarea. La salida de la RN convolucional es el mapa de características (Albawi et al., 2017). Es en este punto donde la RN determina las características más relevantes, se puede decir que los parámetros antes seleccionados por el usuario ahora los determina la RN convolucional (Khan et al., 2018).

La capa de pooling tiene como objetivo reducir la muestra anteriormente procesada, lo que disminuye la carga computacional, utilización de memoria y número de parámetros, extrayendo subsecuencias de una entrada y devolviendo el valor máximo o promedio.

Otra capa interesante que puede tener una RN convolucional es la de batchnormalization. Esta capa normaliza el valor medio y la varianza de la salida de activación. Hay grandes discusiones en torno a las consecuencias estrictas de esta capa (Ioffe & Szegedy, 2015) pero es innegable su capacidad de mejora de la velocidad de convergencia y estabilidad frente a la elección de hiperparámetros o a la inicialización de los pesos.

Utilizando una red convolucional es posible generar análisis de señales de mayor complejidad. El aprendizaje en los coeficientes del kernel sumado a las sucesivas capas como de activación (por ejemplo, de rectificación lineal (ReLU), extracción de promedios o máximos (MaxPooling), las mejoras aportadas por la capa de batchnormalization y las ya conocidas Fully-connected permiten una identificación de patrones más profunda.

Se propuso una red conformada por una capa Convolucional, ver Fig. 195 y se fueron modificando los hiperparámetros para su funcionamiento óptimo. Se eligieron 10 filtros con una dimensión del kernel de 5x1 ya que es una señal unidimensional. Se configuro dependiendo el caso, una cantidad máxima de epochs de 30, 60 o 90. A continuación, se colocó una capa ReLU seguida de una capa MaxPool que extrae el máximo valor dentro 5 lugares y se desplaza un stride de 5, finalmente dos capas fully-Conected de 10 neuronas para obtener el valor regresivo deseado. El código del armado de las capas queda en Matlab de la siguiente manera:

```
layers = [ ...
    imageInputLayer([2000 1 1])
    convolution2dLayer([5 1],10)
    reluLayer
    maxPooling2dLayer([5 1],'Stride',5)
    fullyConnectedLayer(10)
```

```

fullyConnectedLayer(10)
fullyConnectedLayer(1)
regressionLayer];

```

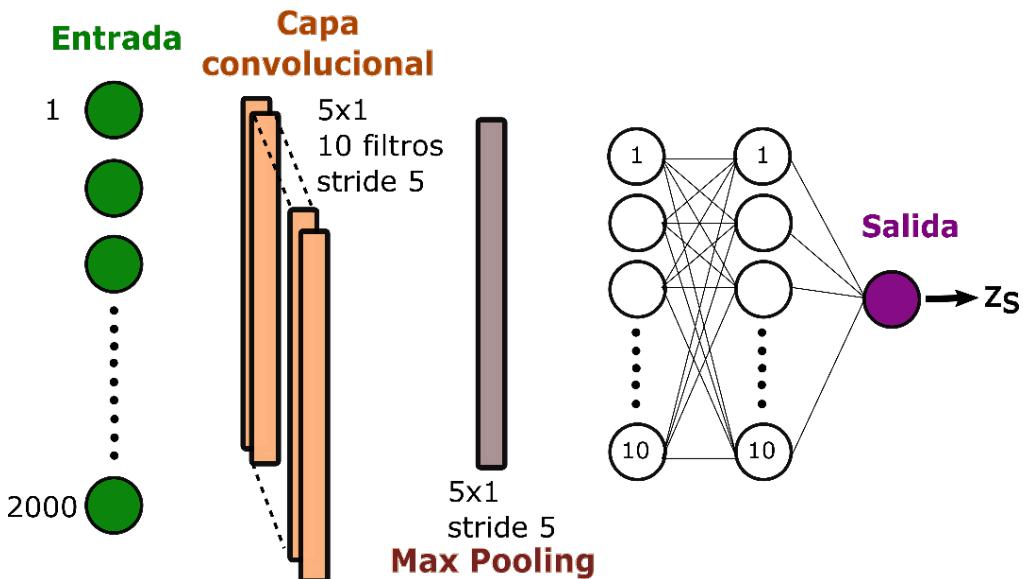


Fig. 195 -Esquema de la red neuronal que contiene una única capa convolucional, seguido de una capa max Pooling y dos capas densamente conectadas de 10 neuronas.

A esta red se la entreno con la señal de la ecuación (11.22) , para poder comparar los resultados con las RNs anteriores, donde se obtuvo un $RMSE = 1.5 \mu\text{m}$ y $RSE_{MAX} = 6.9 \mu\text{m}$, se usó el conjunto de 30000 señales, 60 epochs para el entrenamiento. En la Fig. 196 se puede observar la curva de las frecuencias estimadas y de las frecuencias utilizadas, la precisión obtenida es mejor que para la red anterior propuesta y necesita de menos cantidad de señales para su entrenamiento evidenciando su mayor capacidad de aprendizaje. El error está por debajo de los valores típicos y tolerables de LCI.

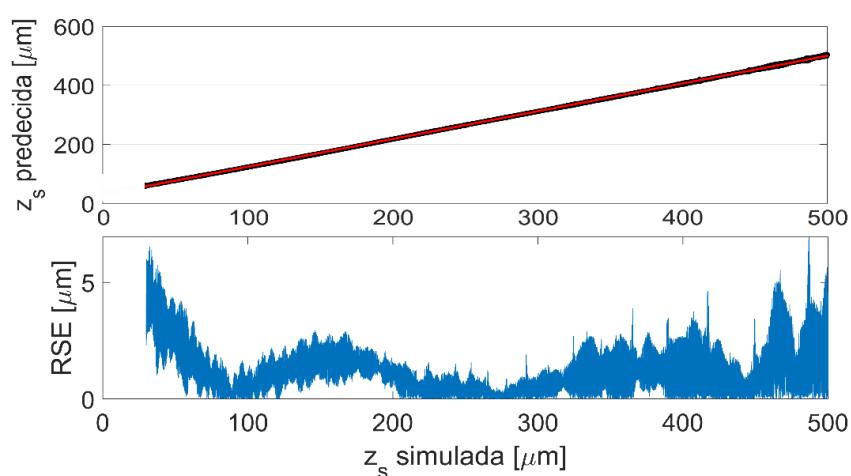


Fig. 196 - Predicción y error absoluto de la red convolucional para todos los valores de z_s para señales cosenoidales con presencia de fases distintas.

Claramente presenta una mejoría con respecto a los casos utilizados anteriormente, donde además el tiempo utilizado para su entrenamiento fue de 18 minutos, cuestión de interés ya que permite entrenar a la red rápidamente si se cambian parámetros del sistema de medición, como por ejemplo el propio espectrómetro que se utiliza en la adquisición. El tiempo de ejecución de la red es de 2.3 ms.

Si bajo las mismas condiciones anteriores se utilizan señales no normalizadas, es decir con amplitudes variadas emulando por ejemplo las distintas reflectividades de la muestra a analizar:

$$i(k) = A \cos(kDCO + \varphi) \quad (11.24)$$

Donde para cada DCO simulada se tomó un valor de A aleatorio comprendido entre 0 y 1, se obtienen las siguientes predicciones:

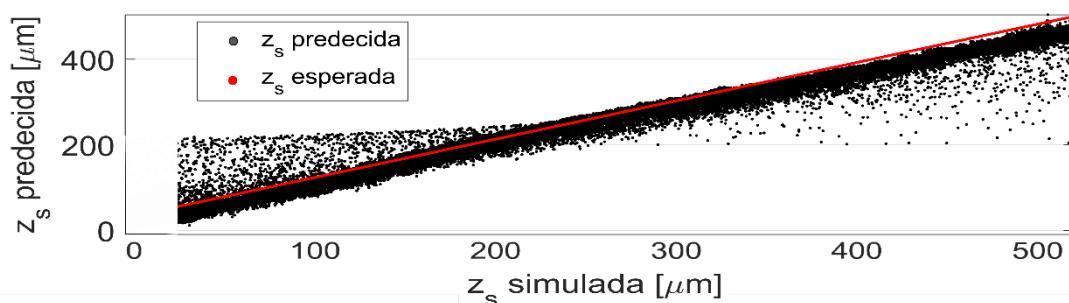


Fig. 197 - Predicción para señales cosenoidales con presencia de fase y amplitudes distintas con red convolucional.

Donde el valor de $RMSE = 37 \mu\text{m}$, lo que indica un incremento casi de un orden del error como se puede observar en la Fig. 197. Este comportamiento se evidenció independientemente de que señal era, siempre que las señales no estaban normalizadas la precisión de la estimación empeoraba. Esto nos permite de manera práctica, postular que es mejor que las señales que se utilicen se encuentren normalizadas. La normalización permite obtener una mejor aproximación sobre el mismo modelo de RN.

Volvamos a la situación donde se pretende analizar la señal de LCI con envolvente, ecuación (11.23), en este caso se utilizaron 30000 señales normalizadas, y la misma RN convolucional se la entrenó utilizando los mismos parámetros. Para el caso en que se tomaron 60 epochs se obtuvo $RMSE = 2.5 \mu\text{m}$ y un $RMS_{max} = 7.5 \mu\text{m}$, un tiempo de entrenamiento de 18 minutos y un tiempo de ejecución de 2.4 ms. En la Fig. 198 se muestran los valores de frecuencia esperados y los valores de frecuencias simulados y la curva de error estimados para esta RN. Los resultados obtenidos son altamente positivos ya que se encuentran por debajo de la resolución de la técnica de FD-LCI.,

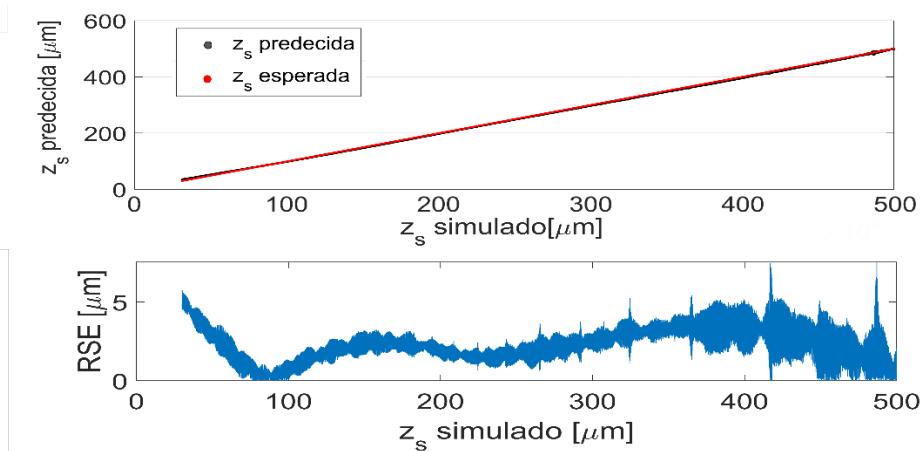


Fig. 198 - Predicciones y error absoluto de la red convolucional para señales.

Bajo estas condiciones se puede acortar el preprocessamiento nuevamente, Fig. 199, dejando solo tres bloques de procesamiento que son el remuestreo, la eliminación de continua y la normalización necesaria.

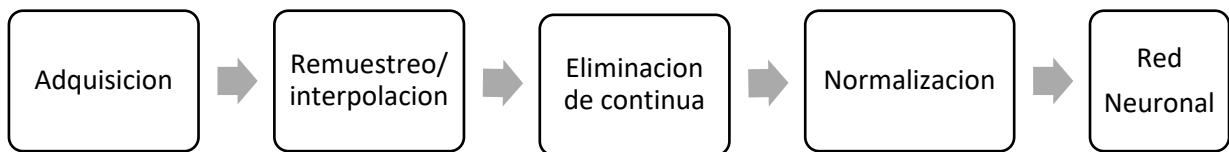


Fig. 199 - Esquema de las etapas de preprocessado, Notar que empleando una red neuronal de mayor capacidad de aprendizaje se puede evitar también la etapa de determinación de la envolvente.

Esta vez se utilizaron las señales simuladas a partir de la ecuación (11.25), donde se contemplan todas las características típicas de una señal de perfilometría sin procesamiento previo, es decir con la presencia de una componente continua A_{DC} y con la envolvente espectral $s(k)$. Recordamos que para cada valor de DCO se tomaron valores aleatorios para A_{DC} , A y φ :

$$i(k) = s(k)[A_{DC} + A \cos(kDCO + \varphi)] \quad (11.25)$$

No se obtuvieron buenos resultados al entrenar la red con el diseño mostrado en la Fig. 195 para este tipo de señales, ya que no es simplemente agregar un valor continuo A_{DC} (notar que la capa ImageInputLayer suele extraer el valor medio de la señal) si no que ese valor esta modulado para cada valor de k por $s(k)$. Por lo tanto, se probaron nuevos diseños de RN. Se fue variando la cantidad de capas convolucionales, la cantidad de filtros por capa como su tamaño. Se buscaron configuración donde el número de filtros aumentaba por cada capa convolucional agregada y otras donde el número de filtros decrecía. Análogamente, se probó con tamaños de los filtros grandes (50×1) en la/s primera/s capas y también con filtros pequeños para todas las capas (5×1 , 3×1 , etc.). También se fue variando el paso del stride de los filtros y agrego “padding” ya que estos repercuten fuertemente en la dimensión de la salida de cada capa. Además, se agregaron capas de Batchnormalization y

diferentes capas de activación después de cada capa convolucional, se fue modificando la cantidad de capas densamente conectadas y su número de neuronas en función de la dimensión de la salida de la última capa convolucional y/o de extracción (max pooling). Vale la pena notar el aumento significativo del tiempo de entrenamiento al tomar más cantidad de capas convolucionales con gran cantidad de filtros y pasos de stride muy pequeños.

El diseño de red que mejor precisión logró fue para una RN de más de 40 capas. Esta configuración posee 7 capas convolucionales donde la cantidad de filtros fue duplicándose, con un tamaño de 2x1 para la primera capa, seguido de 4x1 para las intermedias y de 3x1 para las últimas dos, el paso de stride fue de 2x1 para la primera capa convolucional y de 3x1 para las restantes. Entre capa y capa se tomó la función de activación de tangente hiperbólica, se usó también etapas de batch normalization y de dropout. Finalmente se colocaron capas densamente conectadas de 5000-5000-1000-1000-500-500-250-64-32-16-8-1. El código de la red quedó de la siguiente manera:

```
layers = [ ...
    imageInputLayer([2000 1 1])
    convolution2dLayer([2 1],8, 'Stride',[2 1], 'padding', 'same')
    batchNormalizationLayer
    tanhLayer
    convolution2dLayer([4 1],16, 'Stride',[3 1], 'padding', 'same')
    batchNormalizationLayer
    tanhLayer
    convolution2dLayer([4 1],32, 'Stride',[3 1], 'padding', 'same')
    batchNormalizationLayer
    tanhLayer
    convolution2dLayer([4 1],64, 'Stride',[3 1], 'padding', 'same')
    batchNormalizationLayer
    tanhLayer
    convolution2dLayer([4 1],128, 'Stride',[3 1], 'padding', 'same')
    batchNormalizationLayer
    tanhLayer
    convolution2dLayer([3 1],256, 'Stride',[3 1], 'padding', 'same')
    batchNormalizationLayer
    tanhLayer
    convolution2dLayer([3 1],512, 'Stride',[3 1], 'padding', 'same')
    batchNormalizationLayer
    tanhLayer
    dropout(0.5)
    fullyConnectedLayer(5000)
```

```

tanhLayer
fullyConnectedLayer(5000)
tanhLayer
fullyConnectedLayer(1000)
tanhLayer
fullyConnectedLayer(1000)
tanhLayer
fullyConnectedLayer(500)
tanhLayer
fullyConnectedLayer(500)
tanhLayer
fullyConnectedLayer(250)
tanhLayer
fullyConnectedLayer(64)
tanhLayer
fullyConnectedLayer(32)
tanhLayer
fullyConnectedLayer(16)
tanhLayer
fullyConnectedLayer(8)
tanhLayer
fullyConnectedLayer(1)
tanhLayer
regressionLayer];

```

Se uso para el entrenamiento el conjunto de señales de 100.000 señales, la función de optimización del sgdm y se obtuvo un RMSE = 3.3 μm y un RSE_{max} = 104.4 μm , un tiempo de ejecución de 18.3 ms y un tiempo de entrenamiento de 1218 minutos. Como se puede observar en la figura 27, se obtienen en general valores muy próximos a los esperados, salvo por 3 valores que se obtienen predicciones con un alto error y nuevamente se observa un incremento continuo del error para el extremo de mayores z_s pero que se encuentran dentro del error aceptable. Si bien la RN tiene un óptimo desempeño para la mayoría de los valores de DCO existe una muy baja probabilidad de predicción del z_s con gran error.

En general, cuando se realizan una perfilometría con la técnica de LCI, una vez tomadas las mediciones se observa si hay puntos que se encuentran ampliamente fuera de la tendencia de la muestra y de manera aislada, de ser así pueden ser descartados ya que no representan información fiel de la superficie de la muestra. Esto sucedería para el caso de esta RN al predecir uno de los puntos de mayor error. Al tener una gran variación respecto de los valores de su alrededor, pueden no ser tenidos en cuenta para el análisis.

Por otro lado, se observa como la red incrementa considerablemente su estructura, como así también los tiempos tanto para entrenamiento como de ejecución. En especial el tiempo de ejecución aumenta un orden de magnitud haciendo que pierda sentido reemplazar la etapa de preprocesado de eliminación de continua por esta RN.

Se buscó aumentar la información (data augmentation) espejando las señales y/o se tomaron hasta 500.000 señales con diferentes DCO, pero esto no sirvió para corregir esas pequeñas estimaciones con alto grado de error.

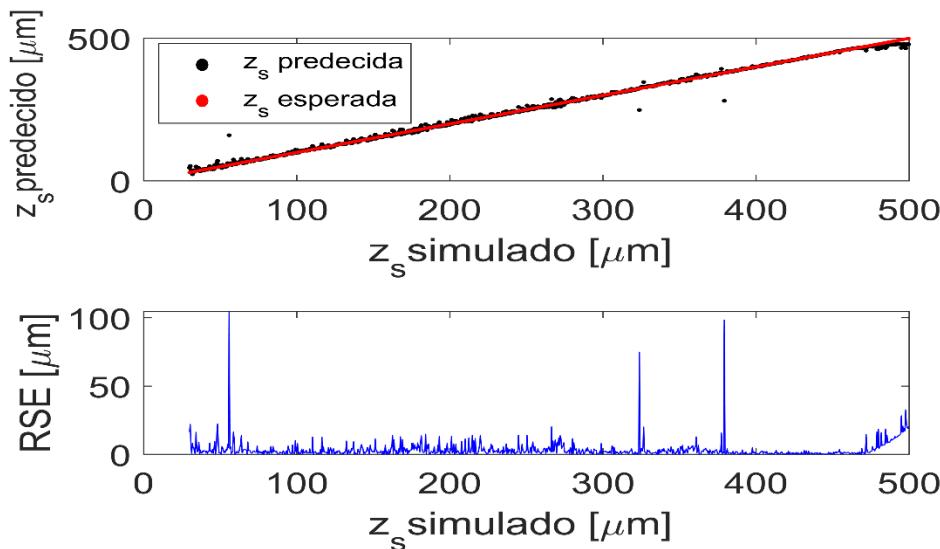


Fig. 200 - Predicciones y error absoluto de la red con 7 capas convolucional para señales típicas de LCI

En la Tabla II se recopilaron los datos de las distintas señales analizadas a lo largo de capítulo, considerando la cantidad de señales utilizadas para el entrenamiento, el diseño de las diversas redes, la precisión y los tiempos tanto de entrenamiento como de ejecución para poder evaluar de una manera rápida el desempeño de las redes.

En general dada un diseño de RN establecido una manera de aumentar la precisión es incrementando el número de señales con las que se entrena y también aumentando el número de epochs. Esto obviamente posee un límite que esta dado por la capacidad de aprendizaje de la red como se mostró a lo largo del capítulo. Si se duplica la cantidad de epochs el tiempo de entrenamiento también se duplica mientras que si se duplica la cantidad de señales el tiempo de entrenamiento crece menos que el doble.

Tabla XVIII – Resumen de los resultados de las redes y señales de mayor precisión. Se evidencian los tiempos tanto de entrenamiento como de ejecución.

Señal	Cantidad de señales	epoch	Red Neuronal	RMSE [μm]	RSE _{max} [μm]	Tiempo entrenamiento [minutos]	Tiempo ejecución [ms]
$\cos(kDCO)$	100.000	60	FCN 2-2-3	2.3	14.8	7	2.0
$\cos(kDCO + \phi)$	100.000	90	FCN 100-500-250-125-	2.2	16.5	27	4.8
$\cos(kDCO + \phi)$	30.000	60	Conv-Relu-MaxPool-FCN 10-10	1.5	6.9	18	2.3
$s(k) \cos(kDCO + \phi)$	30.000	60	Conv-Relu-MaxPool-FCN 10-10	2.5	7.5	18	2.4
$s(k)[A_{DC} + A \cos(kDCO + \varphi)]$	100.000	60	7 Conv	6.3	104.3	1218	18.4

Para estimar la frecuencia de señales oscilante simples la mejor opción es usar una RN FC 2-2-3 ya que se puede obtener una muy buena precisión con pocos minutos de entrenamiento y el tiempo de ejecución es el más bajo respecto a las otras redes. Si la señal a analizar tiene mayor grado de complejidad (presencia de desfasaje, señal continua, variación en la amplitud) conviene utilizar redes convolucionales ya que debido a su gran capacidad de aprendizaje requiere de menor cantidad de señales, epochs y tiempo para su entrenamiento, logrando una alta precisión y con bajo tiempo de ejecución. En particular para analizar las señales de LCI lo mejor es implementar la opción hibrida entre redes neuronales y preprocesado que se observa en la Figura 26 ya que se logran precisión por debajo del error característico de LCI y se reduce el tiempo de cómputo.

Conclusiones

En este capítulo se estudió el uso de diferentes diseños de RN para estimar frecuencias de distintos tipos de señales oscilantes analizando su precisión y tiempos característicos, tanto de entrenamiento como de ejecución. Se mostró que el uso de redes neuronales del tipo densamente conectadas como convolucionales son una alternativa interesante para reducir el preprocesado necesario y reemplazar el uso de la transformada de Fourier para calcular la información de interés en la técnica de FD-LCI.

Se concluyó que la mejor opción para estimar la DCO de una señal de LCI es mantener las etapas de preprocesado de eliminación de continua y normalización, Fig. 199 , y usar una red neuronal con una capa convolucional. Comparando este esquema con el de la Fig. 176 y teniendo en cuenta los tiempos de ejecución de las Tabla XVII y la Tabla XVIII, se observa que teniendo ya entrenada la red neuronal se puede estimar la frecuencia de una señal de aproximadamente 2 ms más rápido. En caso de otras aplicaciones donde, la señal oscillatoria sea más sencilla (como senos o cosenos) usar redes densamente conectadas es una muy buena opción.

Se continuó profundizando en el estudio sobre diferentes redes neuronales para el análisis de frecuencias y se aplicó para señales ópticas de interferencia. Si bien existen algunos trabajos recientes que usan redes densamente conectadas para estimar frecuencias de señales senoidales con presencia de ruido (Sajedian & Rho, 2019) (Almayyali & Hussain, 2021) y otros que usan redes neuronales convolucionales para estimar Modulaciones lineales de frecuencia (Chen et al., 2019). Es un tema que aún falta mucho por estudiar y más aun buscando nuevos ámbitos donde aplicarlas.

Los resultados obtenidos en esta sección se basan entorno a señales de interferencia de baja coherencia, pero fácilmente se pueden implementar para otro ámbito donde se desea analizar la frecuencia de señales similares o inclusive tomar la estructura de las redes neuronales propuestas y modificarlas para optimizar su rendimiento en la aplicación deseada.

Algunas de las ideas a seguir trabajando en el futuro son: continuar trabajando en el diseño de RN para que permitan estimar señales que posean más de una frecuencia presente. Esto permitirá extender el uso del aprendizaje profundo (deep learning) para un gran número de nuevas aplicaciones, puntualmente dentro de la interferencia de baja coherencia permitirá realizar tomografías ya que se podrá identificar información de las interfaces dentro de material. Además, gracias al entrenamiento que requiere la red es altamente probable que se pueda extraer la etapa de remuestreo/interpolación, que es muy importante en caso de realizar la fft, pero no si se desea usar una red neuronal, permitiendo reducir aún más los tiempos de estimación de frecuencia. Por último, nos gustaría explotar la característica de personalización de las redes neuronales. Dado que es necesario su entrenamiento, personalizar su aprendizaje para que identifique más fácilmente características puntuales de la muestra de interés como así también que permite minimizar algún artefacto asociado al sistema de detección.

Referencias

- Albawi, S., Mohammed, T. A., & Al-Zawi, S. (2017). Understanding of a Convolutional Neural Network. *2017 International Conference on Engineering and Technology (ICET) IEEE*. <https://doi.org/10.1109/ICEngTechnol.2017.8308186>
- Ali, M., & Parlapalli, R. (2010). Signal processing overview of optical coherence tomography systems for medical imaging. *Texas Instruments, June, June, 1–22*. <http://www.tij.co.jp/jp/lit/wp/sprabb9/sprabb9.pdf>
- Almayyali, H. R., & Hussain, Z. M. (2021). Deep learning versus spectral techniques for frequency estimation of single tones: Reduced complexity for software-defined radio and iot sensor communications. *Sensors, 21(8)*. <https://doi.org/10.3390/s21082729>
- Bamler, R. (1991). Doppler frequency estimation and the Cramer-Rao bound. *IEEE Transactions on Geoscience and Remote Sensing, 29(3)*, 385–390. <https://doi.org/10.1109/36.79429>
- Belega, D., & Dallet, D. (2008). Frequency estimation via weighted multipoint interpolated DFT. *IET Science, Measurement and Technology, 2(1)*, 1–8. <https://doi.org/10.1049/iet-smt:20070022>
- Cerrotta, S., Morel, E. N., & Torga, J. R. (2015). Scanning Optical Coherence Tomography Applied to the Characterization of Surfaces and Coatings. *Procedia Materials Science, 9*, 142–149. <https://doi.org/10.1016/j.mspro.2015.04.018>
- Chen, X., Jiang, Q., Su, N., Chen, B., & Guan, J. (2019). LFM signal detection and estimation based on deep convolutional neural network. *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference, APSIPA ASC 2019, November, 753–758*. <https://doi.org/10.1109/APSIPAASC47483.2019.9023016>
- Classen, F., & Meyr, H. (1994). Frequency synchronization algorithms for OFDM systems suitable for communication over frequency selective fading channels. *IEEE Vehicular Technology Conference, 3*, 1655–1659. <https://doi.org/10.1109/vetec.1994.345377>
- Deng, L. (2008). Expanding the scope of signal processing. *IEEE Signal Processing Magazine, 25(3)*. <https://doi.org/10.1109/MSP.2008.920380>
- Dorrer, C., Belabas, N., Likforman, J.-P., & Joffre, M. (2000). Spectral resolution and sampling issues in Fourier-transform spectral interferometry. *Journal of the Optical Society of America B, 17(10)*, 1795. <https://doi.org/10.1364/JOSAB.17.001795>
- Dufour, M. (2006). Inspection of hard-to-reach industrial parts using small-diameter probes. *SPIE Newsroom, January, 2–5*. <https://doi.org/10.1117/2.1200610.0467>
- Gao, X., Jin, S., Wen, C. K., & Li, G. Y. (2018). ComNet: Combination of Deep Learning and Expert Knowledge in OFDM Receivers. *IEEE Communications Letters, 22(12)*, 2627–2630. <https://doi.org/10.1109/LCOMM.2018.2877965>
- Grulkowski, I., Liu, J. J., Potsaid, B., Jayaraman, V., Jiang, J., Fujimoto, J. G., & Cable, A. E. (2013). High-precision, high-accuracy ultralong-range swept-source optical coherence tomography using vertical cavity surface emitting laser light source. *Optics Letters, 38(5)*, 673–675.
- Huang, D., Swanson, E. A., Lin, C. P., Schuman, J. S., Stinson, W. G., Chang, W., Hee, M. R., Flotte, T., Gregory, K., Puliafito, C. A., & Fujimoto, J. G. (1991). Optical Coherence Tomography. *Science, 254(5035)*, 1178–1181. <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4638169/>
- Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *32nd International Conference on Machine Learning, ICML 2015, 1*, 448–456.

- Khan, S., Rahmani, H., Shah, S. A. A., & Bennamoun, M. (2018). A Guide to Convolutional Neural Networks for Computer Vision. In G. Medioni & S. Dickinson (Eds.), *Synthesis Lectures on Computer Vision* (Vol. 8, Issue 1). morgan&claypool. <https://doi.org/10.2200/s00822ed1v01y201712cov015>
- Kiranyaz, S., Avci, O., Abdeljaber, O., Ince, T., Gabbouj, M., & Inman, D. J. (2021). 1D convolutional neural networks and applications: A survey. *Mechanical Systems and Signal Processing*, 151, 107398. <https://doi.org/10.1016/j.ymssp.2020.107398>
- Kiranyaz, S., Ince, T., Abdeljaber, O., Avci, O., & Gabbouj, M. (2019). 1-D Convolutional Neural Networks for Signal Processing Applications. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings, 2019-May*, 8360–8364. <https://doi.org/10.1109/ICASSP.2019.8682194>
- Lai, L. L., Tse, C. T., Chan, W. L., & So, A. T. P. (1999). Real-time frequency and harmonic evaluation using artificial neural networks. *IEEE Transactions on Power Delivery*, 14(1), 52–57. <https://doi.org/10.1109/61.736681>
- LeCun, Y., Chopra, S., Ranzato, M. A., & Huang, F. J. (2007). Energy-based models in document recognition and computer vision. *Proceedings of the International Conference on Document Analysis and Recognition, ICDAR*, 1(lcdar), 337–341. <https://doi.org/10.1109/ICDAR.2007.4378728>
- Liu, L., McLernon, D., Ghogho, M., Hu, W., & Huang, J. (2012). Ballistic missile detection via micro-Doppler frequency estimation from radar return. *Digital Signal Processing: A Review Journal*, 22(1), 87–95. <https://doi.org/10.1016/j.dsp.2011.10.009>
- Martínez-Montejano, R. C., Castillo Meraz, R., Lozoya-Ponce, R. E., Campos-Cantón, I., Martínez-Montejano, M. F., & Lozoya Ponce, R. O. (2014). Phase locked loop based on adaptive observer. *International Review of Electrical Engineering*, 9(1), 244–248. <https://doi.org/10.15866/iree.v9i1.221>
- Morel, E. N., Russo, N. A., Torga, J. R., & Duchowicz, R. (2016). Interferometric system based on swept source-optical coherence tomography scheme applied to the measurement of distances of industrial interest. *Optical Engineering*, 55(1), 014105-1/7. <https://doi.org/10.1117/1.OE.55.1.014105>
- Orović, I., Stanković, S., Thayaparan, T., & Stanković, L. J. (2010). Multiwindow S-method for instantaneous frequency estimation and its application in radar signal analysis. *IET Signal Processing*, 4(4), 363–370. <https://doi.org/10.1049/iet-spr.2009.0059>
- Rosemarie, V. (2008). Discrete Fourier Transform computation using neural networks. *Proceedings - 2008 International Conference on Computational Intelligence and Security, CIS 2008*, 1, 120–123. <https://doi.org/10.1109/CIS.2008.36>
- Routray, A., Pradhan, A. K., & Rao, K. P. (2002). A novel Kalman filter for frequency estimation of distorted signals in power systems. *IEEE Transactions on Instrumentation and Measurement*, 51(3), 469–479. <https://doi.org/10.1109/TIM.2002.1017717>
- Sajedian, I., & Rho, J. (2019). Accurate and instant frequency estimation from noisy sinusoidal waves by deep learning. *Nano Convergence*, 6(1), 2–6. <https://doi.org/10.1186/s40580-019-0197-y>
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. <https://doi.org/10.1109/CVPR.2015.7298594>
- Tang, Y., & Eliasmith, C. (2010). Deep networks for robust visual recognition. *ICML 2010 - Proceedings, 27th International Conference on Machine Learning*, 1055–1062.

- Walecki, W. J., Lai, K., Pravdivtsev, A., Souchkov, V., Van, P., Azfar, T., Wong, T., Lau, S. H., & Koo, A. (2005). Low-coherence interferometric absolute distance gauge for study of MEMS structures. *Reliability, Packaging, Testing, and Characterization of MEMS/MOEMS IV*, 5716, 182. <https://doi.org/10.1117/12.590013>
- Walecki, W. J., Pravdivtsev, A., Santos II, M., & Koo, A. (2006). High-speed high-accuracy fiber optic low-coherence interferometry for in situ grinding and etching process monitoring. *Interferometry XIII: Applications*, 6293, 62930D. <https://doi.org/10.1117/12.675592>
- Wang, M. R. (n.d.). *Optical Coherence Tomography and Its Non-medical Applications*.
- Wang, T., Wen, C. K., Jin, S., & Li, G. Y. (2019). Deep Learning-Based CSI Feedback Approach for Time-Varying Massive MIMO Channels. *IEEE Wireless Communications Letters*, 8(2), 416–419. <https://doi.org/10.1109/LWC.2018.2874264>
- Xu, S., & Shimodaira, H. (2019). Direct F0 estimation with neural-network-based regression. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH, 2019-Septe*, 1995–1999. <https://doi.org/10.21437/Interspeech.2019-3267>
- Yardibi, T., Li, J., Stoica, P., Xue, M., & Baggeroer, A. B. (2010). Source localization and sensing: a nonparametric iterative adaptive approach based on weighted least squares. *IEEE Transactions on Aerospace and Electronic Systems*, 46(1), 425–443. <https://doi.org/10.1109/TAES.2010.5417172>

Capítulo XII - Redes Neuronales aplicadas a la detección de picos de glucosa después de las comidas en series temporales de pacientes diabéticos tipo II

Hoy en día existen sensores intersticiales que mejoran la calidad de vida de pacientes diabéticos, los picos de glucosa son interesantes de estudiar porque pueden dar información sobre la cantidad de carbohidratos consumidos para un ajuste posterior, que el médico especialista podrá hacer en el tratamiento del paciente. En este capítulo, entremos en redes neuronales para poder detectar estos picos. Analizaremos qué estrategia usar, preparamos los datos y probaremos si la estrategia aplicada funciona como esperamos.

Analizando los datos

Tomaremos datos acumulados por el momento de dos pacientes diabéticos a quienes llamaremos *sujeto_1* y *sujeto_2* y aprovecharemos sus anotaciones para saber cuándo se hizo una ingesta de comida. Ilustraremos la idea tomando un día del *sujeto_1*. Como el sensor toma las muestras cada 15 min y son 24 horas crearemos una variable *t* que maneje el tiempo y obtendremos una gráfica de la serie temporal, Fig. 201.

```
t=0:(24/95):24; % el paso es de 24/95
load('G1-10-2019.mat') % cargamos los datos
plot(t,G)
hold on
grid on
xlabel('tiempo')
ylabel('Glucosa')
```

Sabemos de sus anotaciones que alrededor de las 8:30 hizo un desayuno. Gracias a la bibliografía científica el tiempo aproximado en que se da el pico de glucosa según Daen (2010) es de aproximadamente de 72 ± 23 min y según Chapelot (2007) en 47 ± 3 min. Si bien parecen muy lejanos, tomando el menor valor posible de Dean este resulta ser de 49 min acercándose al de Chapelot y podemos observar en la serie temporal este pico se da aproximadamente a la hora.

```
plot(t(find(t>=8.5 & t<=9.5)), G(find(t>=8.5 & t<=9.5)), 'LineWidth',2)
% Visualizamos la sección que nos interesa prestar atención
```

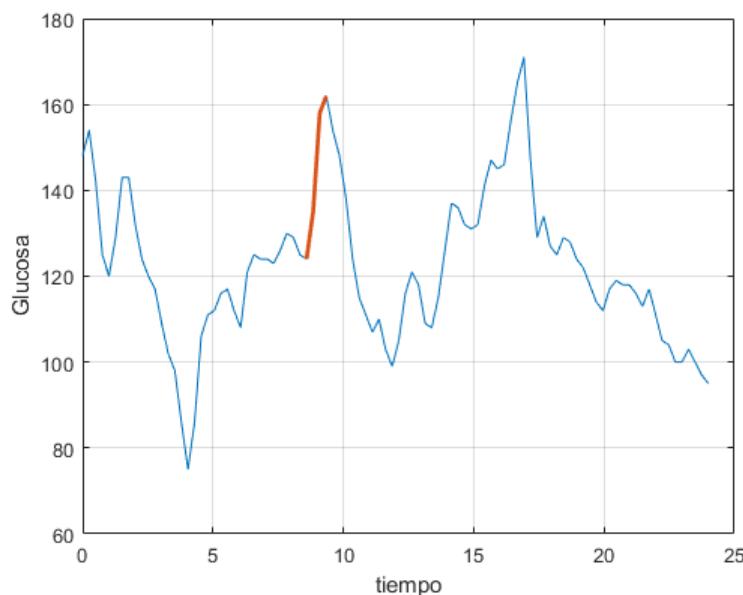


Fig. 201 – Representación de la evolución de la glucosa, en rojo desayuno del sujeto_1.

Si nos centramos en la característica de esa sección de la señal podemos decir que está bien correlacionada y tiene una pendiente que la describe, Fig. 202.

```

figure
ts = t(find(t>=8.5 & t<=9.5))

ts = 1×4
    8.5895    8.8421    9.0947    9.3474

gs = G(find(t>=8.5 & t<=9.5))'

gs = 1×4
    124    135    158    162

plot(ts, gs)
cor=corrcoef(ts, gs); %Se obtiene la correlación
cor(1,2)

ans = 0.9693

valores= polyfit(ts, gs,1);
m=valores(1) %Se obtiene la pendiente que la caracteriza.

m = 54.2292

b=valores(2)

b = -341.6000

```

```

hold on
grid on
plot(ts, m*ts+b)
xlabel('tiempo')
ylabel('Glucosa')
legend(['Ascenso de glucosa'], ['Recta de regresión'])

```

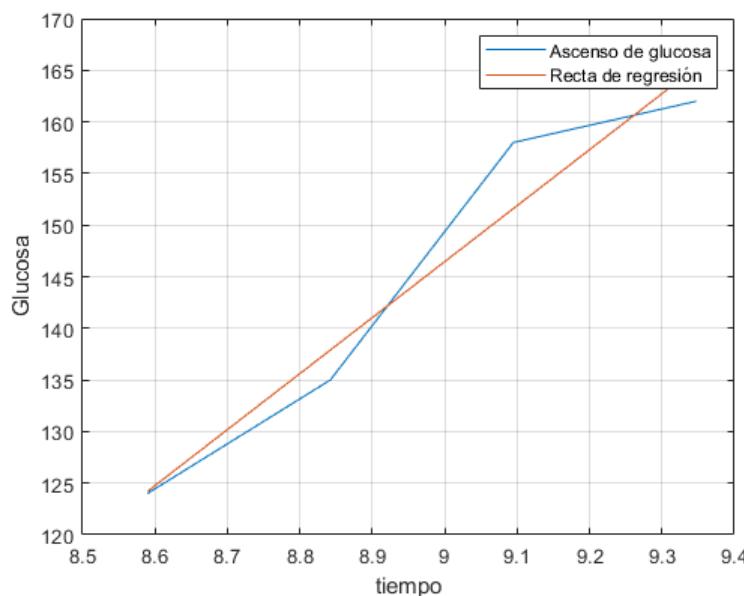


Fig. 202 – Representación del ascenso de la glucosa, en rojo recta de regresión.

Analizando el registro de 10 días del *sujeto_1* tanto de su glucosa como de sus anotaciones, se puede inferir que las comidas llegan al pico máximo aproximadamente en 1h, aunque existen casos en que se registran 1h 15 min o 45 min, que equivalen a 4 puntos de muestra en general y en algunos casos estos pueden rondar en 3 puntos o 5 puntos. Para detectar los intervalos de la ingesta de la comida y el pico máximo que tenga en cuenta las pequeñas variaciones de tiempo descriptas, se realizará una enventanado de 1h con desplazamiento de 15 min. Se identificará con un 0 aquellos valores de correlación y pendiente que no correspondan a la ingesta de comida y con un 1 aquellos que, si lo estén como, se muestra un ejemplo en la Tabla XIX.

Tabla XIX – Ejemplo de tabla de ingestas

Correlación	Pendiente	Comida: Sí/No
-0,8321	-2,4	0
-0,717	-2,0	0
0,6255	2,4	0
0,8752	4,8	0
0,9439	5,6	0
0,7661	7,2	0
0,8706	18,4	1
0,9531	23,6	1
0,4821	10,4	0

Que gráficamente para la ingesta de comida se vería así, Fig. 203:

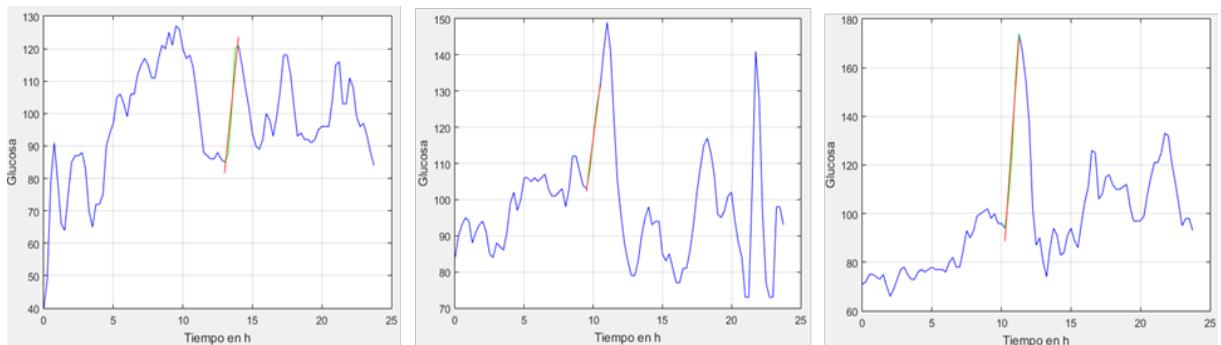


Fig. 203 – Ejemplo de ingesta de comida (a) corr= 0.95 m=42, (b) corr=0.99 m=29.6 (c) corr=0.98 m=83.2

Para la no ingesta Fig. 204:

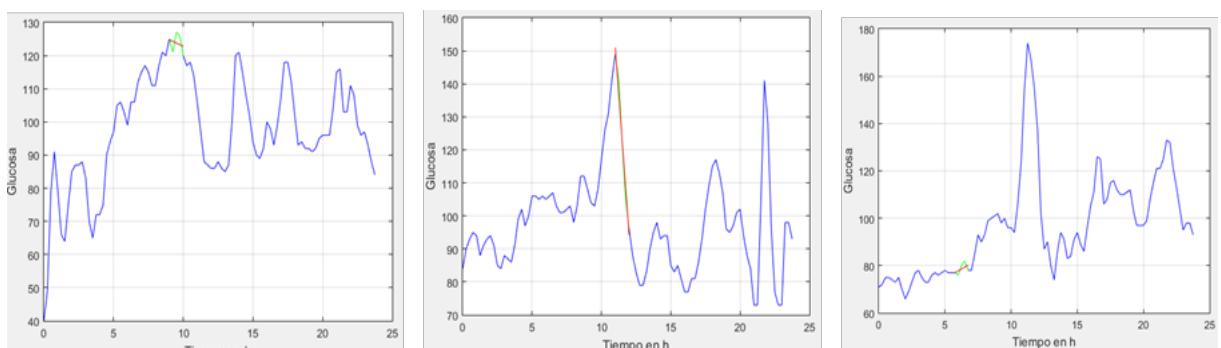


Fig. 204 – Ejemplo de no ingesta de comida (a) corr= -0.95 m=-0.3, (b) corr=-0.99 m=-56.8 (c) corr=0.52 m=3.2

Entrenando la red

Para entrenar la misma deberemos generar la entrada, en este caso tendrá dos (la correlación y la pendiente) y por separado la salida binaria correspondiente (1 - Ingesta, 0 - No ingesta), para ello repasaremos todas las mediciones del *sujeto_1*, realizaremos una ventana que contenga cuatro puntos y se mueva un paso a la vez, esto estaría generando pequeños intervalos en donde indicaremos si corresponde a una ingesta o no en ese intervalo siguiendo las anotaciones del *sujeto_1*. Para que esta tarea sea un poco más amena se utilizará el siguiente código, que simplemente nos ayuda en la tarea:

```

clc
load('G.mat')
tam=size(G);
t=G(:,1); %Se obtiene el tamaño de la serie
            %La primera linea contiene el tiempo
t=t';
intrain=zeros((tam(2)-1)*92, 2); %Se crea la variable que contendrá los datos
de entrada [correlación, pendiente]
outtrain=zeros(1,(tam(2)-1)*92); %Se crea la variable que contendrá la salida
1- Ingesta, 0- No Ingesta
k=1;
for i=1:tim(2) %Se recorre la serie
    disp(i)
    if i~1
        for j=1:92
            clf
            plot(G(:,1), G(:,i), 'b');%Se grafica la serie correspondiente al día
completo
            xlabel('Tiempo en h');
            ylabel('Glucosa');
            hold on
            plot(G(:,1), G(:,i), 'b');
            [intrain(k, 1), intrain(k, 2) b]=Gprep(G(j:j+4,1), G(j:j+4,i));
            %Se obtiene la correlación y la pendiente de la ventana
            plot(G(j:j+4,1), G(j:j+4,i), 'g')
            %Se grafica la ventana en verde
            plot(G(j:j+4,1), intrain(k, 2)*G(j:j+4,1)+ b, 'r');
            %Se grafica la recta regresión en rojo
            grid on
            disp(intrain(k,1))
            %Se muestra la correlación
            disp(intrain(k,2))
            %Se muestra la pendiente
            outrain(k)=input('Ingrese 1 pos si, 0 por no...');

            %Se pide confirmación de ingesta y se guarda la salida
            k=k+1;
        end
    end
end
outtrain=outtrain';

```

Una vez obtenida la misma se entrena la red de manera que tenga la configuración mostrada en la Fig. 205, para ello vamos a usar las herramientas que nos brinda Matlab ejecutando

```
nprtool
```

Ingresamos las variables *intrain* y *outtrain*, seleccionamos el porcentaje de entrenamiento, validación y testeо por defecto, definimos las capas ocultas, se tomaron 20 neuronas. Elegimos el algoritmo *Levenberg-Marquart* y presionamos el botón de Train.

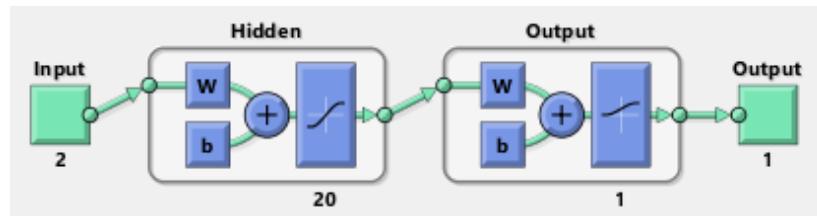


Fig. 205 – Configuración de la red.

En las Fig. 206 se muestran las matrices de confusión obtenidas, y en la Fig. 207 se muestran las curvas ROC (Característica Operativa del Receptor).

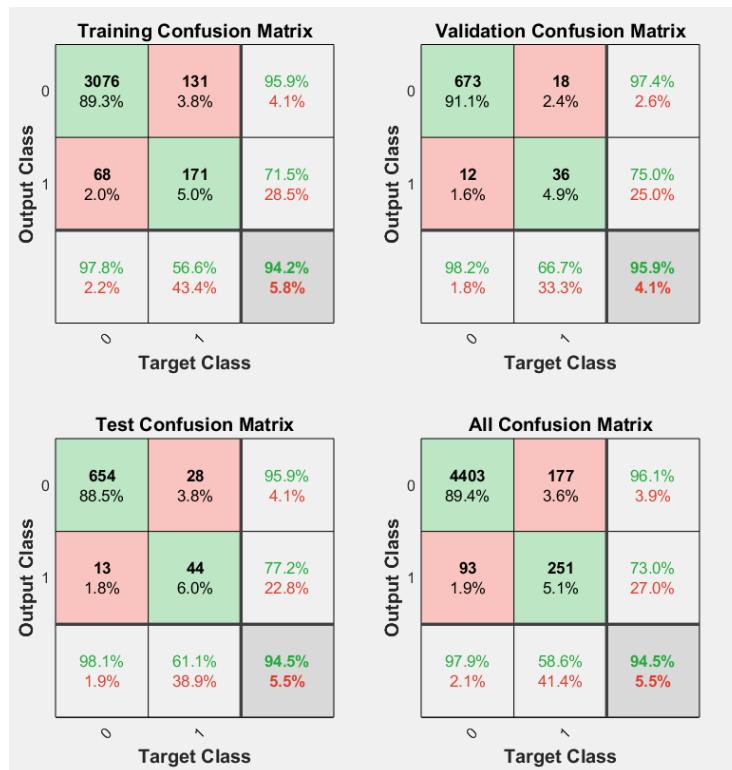


Fig. 206 – Matrices de confusión de red neuronal utilizada para la detección de picos de glucosa

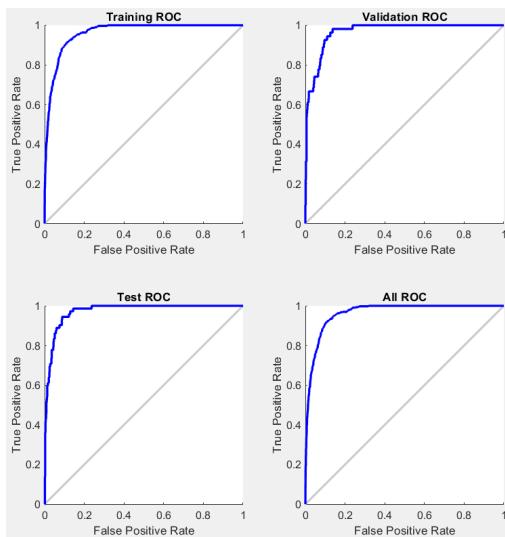


Fig. 207 – Curvas ROC de red neuronal utilizada para la detección de picos de glucosa

Probando la red

Teniendo en cuenta que existen falsos positivos y negativos, retomaremos la estrategia del enventanado y se verificará si se detecta la ingesta de comida del *sujeto_2*.

```
% Análisis de existencia de glucosa mediada por comida con carbohidratos
% Se recorre la serie temporal mediante un enventanado de 1h equivalente
% a 4 muestras

for j=2:14
    figure(1)
    k=0;
    u=1;
    inter=zeros(5,4);
    aux=zeros(5,4);
    ms=0;
    for i=1:92
        figure(1)
        clf
        plot(G(:,1), G(:,j), 'b');
        hold on
        grid on
        %Se recorre la serie cada 15 min en una ventana de 1h
        [c m b]=Gprep(G(i:i+4,1), G(i:i+4,j));
        plot(G(i:i+4,1),G(i:i+4,j), 'g')
        in=[c;m];
        %Se carga la red
        a=sim(net_JV, in);
```

```

pause(0.1)
ms=[ms m];
if round(a)== 1
    plot(G(i:i+4,1),G(i:i+4,j),'y');
    xlabel('Tiempo en horas')
    ylabel('Glucosa')
    pause(0.3)
    text(G(i,1), G(i, j)-10, 'Comida-carbohidratos');
%Se cuenta la cantidad que de ventanas en recorrido que dio positivo como
comida
k=k+1;
aux(k,1)=G(i,j);
aux(k,2)=G(i+4,j);
aux(k,3)=G(i,1);
aux(k,4)=G(i+4,1);
else
    figure(1)
    Intervalo ='[ ';
%Se busca el mínimo local y el máx local de la ingesta
switch k
    case 1
        inter(u,1)=aux(1,1);
        inter(u,2)=aux(1,2);
        inter(u,3)=aux(1,3);
        inter(u,4)=aux(1,4);
        msgG(G,inter,u,j,Intervalo);

        input('Presione una tecla para continuar... ');
        u=u+1;

    case 2
        inter(u,1)=min(aux(1:2,1));
        inter(u,2)=max(aux(1:2,2));
        inter(u,3)= aux(min(find(aux(1:2,1)==inter(u,1))),3);
        inter(u,4)= aux(max(find(aux(1:2,2)==inter(u,2))),4);
        msgG(G,inter,u,j,Intervalo);
        input('Presione una tecla para continuar... ');
        u=u+1;
    case 3
        inter(u,1)=min(aux(1:3,1));
        inter(u,2)=max(aux(1:3,2));
        inter(u,3)= aux(min(find(aux(1:3,1)==inter(u,1))),3);
        inter(u,4)= aux(max(find(aux(1:3,2)==inter(u,2))),4);
        msgG(G,inter,u,j,Intervalo);
        input('Presione una tecla para continuar... ');
        u=u+1;
    case 4
        inter(u,1)=min(aux(1:4,1));

```

```

inter(u,2)=max(aux(1:4,2));
inter(u,3)= aux(find(aux(1:4,1)==inter(u,1)),3);
inter(u,4)= aux(find(aux(1:4,2)==inter(u,2)),4);
msgG(G,inter,u,j,Intervalo);
input('Presione una tecla para continuar... ');
u=u+1;
case 5
    inter(u,1)=min(aux(:,1));
    inter(u,2)=max(aux(:,2));
    inter(u,3)= aux(find(aux(1:5,1)==inter(u,1)),3);
    inter(u,4)= aux(find(aux(1:5,2)==inter(u,2)),4);
    msgG(G,inter,u,j,Intervalo);
    input('Presione una tecla para continuar... ');
    u=u+1;
end
if k > 0
    k=0;
    aux=zeros(3,2);
end

end
end
end

```

En la Fig. 208 se muestran los resultados obtenidos del análisis de existencia de ingesta de comida con su respectivo pico máximo de glucosa.

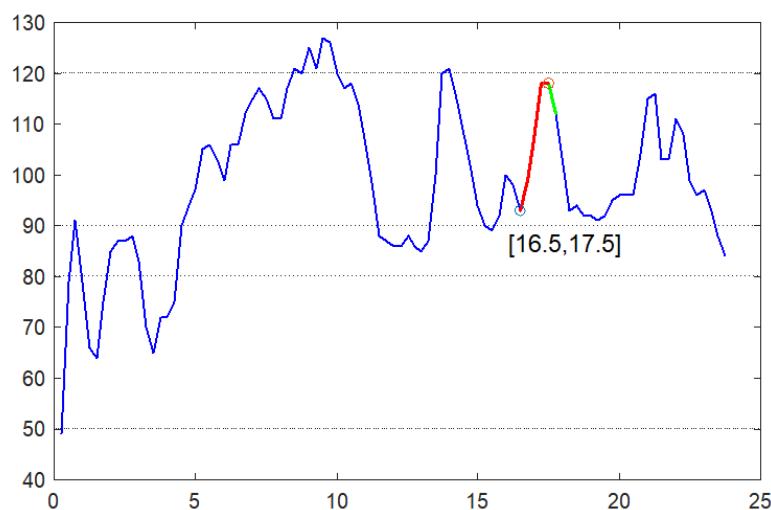


Fig. 208 – Resultados obtenidos para el análisis ingestión de comida con su respectivo pico máximo de glucosa

Referencias

- Bailey, T., Wode, B., Christiansen, M., Klaff, L., & Alva, S. (2015). The Performance and Usability of a Factory-Calibrated. *Diabetes Technology & Therapeutics*, 7(14), 7.
- D. Chapelot, C. M. (2007). Predicting more accurately the overall glucose response to a lunch meal by using the postprandial glucose peak. *Metabolism Clinical and Experimental*, 37-43.
- Simone, O. (2018). A Very Brief Introduction to Machine Learning to Communication System. IEEE.
- S. Daenen, A. S.-G.-B. (2010). Peak-time determination of post-meal glucose excursions. *Diabetes & Metabolism*, 165-169.
- Weinstein, R., Schwartz, S., Razg, S., Jolyun R., B., Thomas A., P., & Geoffrey V., M. (2007). Accuracy of the 5-Day FreeStyle Navigator Continuous Glucose Monitoring System. *Diabetes Care*, 1125-1130.

Anexo I – Lista de ejercicios

Tabla XX – Lista de ejercicios de los capítulos I, II y III (introducción a redes neuronales básicas con algoritmos de entrenamiento y redes RBF)

Ejercicio	Descripción	Plataforma
Ejercicio 1.1	Calculo analítico de Red Neuronal Perceptron	Analítico
Ejercicio 1.2	Regiones de decisión para de Red Neuronal Perceptron	Analítico
Ejercicio 1.3	Cálculo de Neuronal Perceptron para compuerta AND	Analítico y Matlab
Ejercicio 1.4	Red Neuronal tipo Adaline	Matlab
Ejercicio 2.1	Aproximación de funciones mediante el algoritmo de propagación hacia atrás	Analítico
Ejercicio 2.2	Aproximación de funciones con Matlab mediante algoritmo de propagación hacia atrás. Funciones con ruido	Matlab
Ejercicio 2.3	Aproximación de funciones con algoritmo de propagación hacia atrás	Matlab
Ejercicio 2.4	Ajuste de datos mediante una Red Neuronal estática.	
Ejercicio 2.5	Clasificación de datos mediante red neuronal MLP (Perceptron Multi Capa). Matriz de confusión y métricas.	Python
Ejercicio 2.6	Ajuste de datos con las librerías scikit-learn de Python	Python
Ejercicio 3.1	Red neuronal RBF para compuerta XOR	Analítico y Matlab
Ejercicio 3.2	Red neuronal RBF para compuerta XOR con datos ruidosos	Matlab
Ejercicio 3.3	Ejercicio básico de ajuste de datos Red neuronal RBF	Matlab
Ejercicio 3.4	Red neuronal RBF para cálculos de regresión	Python
Ejercicio 3.5	Red neuronal RBF para clasificar datos en 2 dimensiones	Python

Tabla XXI – Lista de ejercicios de los capítulos IV y V (reconocimiento estadístico de patrones, aprendizaje automático y redes SOM)

Ejercicio	Descripción	Plataforma
Ejercicio 4.1	Ejemplos de redes neuronales en Matlab® para ajuste de datos, agrupamiento, reconocimiento de patrones y series temporales	Herramienta nnstart de Matlab
Ejercicio 4.2	Ajuste de datos con redes neuronales estáticas mediante herramienta nftool de Matlab	Matlab
Ejercicio 4.3	Ajuste de datos con redes neuronales estáticas mediante script en Matlab	Matlab
Ejercicio 4.4	Reconocimiento de Patrones: Clasificación de datos con redes neuronales	Matlab
Ejercicio 4.5	Método K-NN	Matlab
Ejercicio 4.6	Ejemplo de Análisis de Componentes Principales (PCA)	Matlab
Ejercicio 4.7	Gráficos de Funciones de densidad de probabilidad	Matlab
Ejercicio 4.8	Gráficos de Funciones de densidad de probabilidad	Analítico y Matlab
Ejercicio 4.9	Ejemplo de Estimación de distribución de probabilidad	Matlab
Ejercicio 4.10	Matriz de confusión en Matlab con 2 clases	Analítico y Matlab
Ejercicio 4.11	Matriz de confusión en Matlab con 4 clases	Matlab
Ejercicio 4.12	Ejemplo simple de red neuronal de reconocimiento de patrones para clasificación de datos en diferentes clases (grupos)	Matlab
Ejercicio 5.1	Redes Neuronales SOM	Matlab
Ejercicio 5.2	Redes Neuronales SOM	Python
Ejercicio 5.3	Redes Neuronales SOM con librería sompy en Python	Python

Tabla XXII – Lista de ejercicios del capítulo VI (redes neuronales dinámicas)

Ejercicio	Descripción	Plataforma
Ejercicio 6.1	Análisis de Respuesta de Red Neuronales dinámicas y estática	Matlab
Ejercicio 6.2	Red Neuronal dinámica ADALINE	Analítico y Matlab
Ejercicio 6.3	Red Neuronal no lineal autoregresiva (NAR)	Matlab
Ejercicio 6.4	Red Neuronal con retardo temporal (Time Delay)	Matlab
Ejercicio 6.5	Red Neuronal dinámica autoregresiva para serie temporal	Matlab
Ejercicio 6.6	Ejemplo de Red Neuronal Recurrente (RNN)	Matlab
Ejercicio 6.7	Predicción de señales temporales mediante una RNN tipo LSTM	Python
Ejercicio 6.8	Traductor de inglés a castellano con modelo de transformación secuencial	Python. Keras Tensorflow
Ejercicio 6.9	Generación de texto con redes neuronales recurrentes (RNN) del tipo GRU	Python. Keras Tensorflow
Ejercicio 6.10	Clasificación de textos mediante el análisis sentimental con RNN bidireccional LSTM	Python. Keras Tensorflow
Ejercicio 6.11	Reconocimiento de acciones para clasificación de videos con red neuronal RNN GRU combinada con CNN	Python. Keras Tensorflow
Ejercicio 6.12	Reconocimiento de acciones para clasificación de videos con red neuronal I3D CNN. Comparación de distintas arquitecturas	Python. Keras Tensorflow
Ejercicio 6.13	Predicción de tramas de video mediante RNN con capas LSTM	Python. Keras Tensorflow
Ejercicio 6.14	Clasificación de vocales con redes LSTM mediante herramienta deepNetworkDesigner.	Matlab deepNetworkDesigner
Ejercicio 6.15	Ejemplo de reconocimiento de voz con espectrogramas, aprendizaje profundo y procesamiento paralelo en Matlab	Matlab deepNetworkDesigner

Tabla XXIII – Lista de ejercicios del capítulo VII (aprendizaje profundo con redes neuronales convolucionales)

Ejercicio	Descripción	Plataforma
Ejercicio 7.1	Ejemplo de convolución continua. Resolución analítica.	Analítico
Ejercicio 7.2	Ejemplo simple de red neuronal “GoogleNet” Recurrente (RNN) en Matlab®, para identificación de imágenes.	Matlab
Ejercicio 7.3	Ejemplo de carga y visualización de redes CNN preentrenadas con herramienta deepNetworkDesigner de Matlab®	Matlab deepNetworkDesigner
Ejercicio 7.4	Mediante la herramienta deepNetworkDesigner de Matlab editar la red CNN SqueezeNet preentrenada, importar datos, entrenar y clasificar	Matlab deepNetworkDesigner
Ejercicio 7.5	Diseño de redes CNN para clasificación de números en Lenguaje Matlab®. con conjunto de datos Digits Dataset.	Matlab
Ejercicio 7.6	Clasificación imágenes webcam con red Alexnet. Lenguaje Matlab®.	Matlab
Ejercicio 7.7	Ejemplo de clasificación de imágenes con red CNN y conjunto de datos CIFAR10.	Python con librerías Pytorch
Ejercicio 7.8	Clasificación de imágenes con red CNN y conjunto de imágenes propias. Lenguaje Python con librerías TensorFlow.	Python con TensorFlow
Ejercicio 7.9	Clasificación de imágenes con red CNN y conjunto de datos CIFAR10. Lenguaje Python con librerías Tensorflow.	Python con TensorFlow
Ejercicio 7.10	Clasificación de imágenes con red CNN y conjunto de datos MNIST. Lenguaje Python con librerías Tensorflow. Comandos para guardar el modelo	Python con TensorFlow
Ejercicio 7.11	Ejemplo simple de detección de elefantes con red CNN pre entrenada ResNet50 y VGG16. Lenguaje Python con librerías Tensorflow.	Python con TensorFlow

Tabla XXIV – Lista de ejercicios del capítulo VIII (aplicaciones en dispositivos móviles y de IoT)

Ejercicio	Descripción	Plataforma
Ejercicio 8.1	Clasificación de imágenes con dígitos mediante el celular	TensorFlow Lite multiplataforma
Ejercicio 8.2	Detección de objetos dentro de una imagen	TensorFlow Lite multiplataforma
Ejercicio 8.3	Mejora en la resolución de imágenes	TensorFlow Lite multiplataforma
Ejercicio 8.4	Ejemplo para crear un modelo propio y entrenarlo con imágenes	TensorFlow Lite multiplataforma
Ejercicio 8.5	Ejemplos agregador de TensorFlow Lite para descargar: Preguntas y respuestas (bert_qa) Clasificación de gestos (gesture_classification) Clasificación de imágenes (image_classification) Segmentación de imágenes (image_segmentation) Modelos personalizados (model_personalization) Estimación de pose (pose_estimation y pose_net) Recomendaciones personales (recommendation) Respuestas a mensajes de chat (smart_reply) Clasificación de sonidos (sound_classification) Comandos de voz (speech_commands) Transferencia de estilos artísticos (style_transfer) Clasificación de texto (text_classification)	TensorFlow Lite multiplataforma

Tabla XXV – Lista de ejercicios de los capítulos IX y X (visión artificial y aplicaciones con métodos combinados)

Ejercicio	Descripción	Plataforma
Ejercicio 9.1	Ejemplo para contar manzanas con función detector de bordes Canny	OpenCV en Python
Ejercicio 9.1	Ejemplo para contar manzanas con función detector de bordes Canny	OpenCV en Python
Ejercicio 9.2	Ejemplo para contar monedas mediante función SimpleBlobDetector() de OpenCV	OpenCV en Python
Ejercicio 9.3	Ejemplo para contar monedas con la función de detección de bordes Canny	OpenCV en Python
Ejercicio 9.4	Ejemplo de detección de rostros en una imagen con Haar Cascade Classifier	OpenCV en Python
Ejercicio 9.5	Ejemplo de detección de rostros en un archivo de video con Haar Cascade Classifier	OpenCV en Python
Ejercicio 9.6	Ejemplo de detección de sonrisas en una imagen con Haar Cascade Classifier	OpenCV en Python
Ejercicio 9.7	Detección de esquinas mediante la función corner Harris de OpenCV	OpenCV en Python
Ejercicio 9.8	Detección de esquinas usando el método Shi-Tomasi con función goodFeaturesToTrack	OpenCV en Python
Ejercicio 9.9	Detección de bordes con webcam y función Canny	
Ejercicio 9.10	Búsqueda y detección de objetos o imágenes dentro de otra imagen	OpenCV en Python
Ejercicio 10.1	Ejemplo de clasificación de muestras de vinos, mediante PCA y LDA, clasificadores y redes neuronales.	Python, librerías sklearn
Ejercicio 10.2	Ejemplo de separación de muestras de flores del conjunto de datos IRIS	Orange
Ejercicio 10.3	Ejemplo de separación de muestras LIBS mediante programa con interfaz gráfica de Orange	Orange
Ejercicio 10.4	Ejemplo de clasificación de muestras con diferentes redes neuronales y métodos estadísticos	Python, librerías sklearn. RapidMiner

Tabla XXVI – Detalles de aplicaciones de los capítulos XI y XII

	Descripción	Plataforma
Capítulo XI	Aplicación avanzada de procesado de señales interferométricas con redes neuronales: Estimando frecuencias	Matlab trainNetwork
Capítulo XII	Redes neuronales para la detección de gluconeogénesis alterada y picos de glucosa.	Matlab

Epílogo

Existen numerosas técnicas de procesamiento de datos para resolver un mismo problema. Tomar decisiones basadas solamente en el procesamiento de datos, sin realizar un análisis previo, muchas veces puede resultar lento y contraproducente. Si el problema está mal planteado se pueden cometer errores graves.

Hay que utilizar la intuición humana, tener un claro entendimiento de los métodos utilizados y evaluar los modelos para plantear correctamente las posibles soluciones.

En un sistema mal diseñado, el procesamiento de datos puede fallar ante pequeños cambios de las entradas, por tal motivo se busca que los modelos tengan buena generalización.

También se debe tener en cuenta que los sistemas de inteligencia artificial deben ser dinámicos y se deben adaptar a los cambios del entorno, modificando rápidamente su funcionamiento.

Resulta necesario conocer distintas técnicas para abarcar adecuadamente el problema. De esta forma se trata de buscar resultados que sean lo más satisfactorios posibles, teniendo en cuenta que los métodos no son infalibles, pero se trata de minimizar el error y bajar la carga de recursos de hardware y la carga computacional.

Se espera que el contenido del libro sea de utilidad al lector para el desarrollo de sus aplicaciones de ingeniería científico tecnológicas. Así también los conceptos de inteligencia artificial de este libro se pueden aplicar en muchas disciplinas tales como gestión de empresas, análisis y decisiones comerciales y financieras, análisis sociales, etc.

Para citar el presente libro por favor incluir los datos completos con número de ISBN.

Contenidos

Prólogo	1
Capítulo I - Introducción a Redes Neuronales.....	3
Introducción a la inteligencia artificial	3
Introducción a Redes Neuronales	4
Entradas y capas.....	9
Breve historia de las redes neuronales.....	11
Clasificación de Redes Neuronales.....	12
Red Neuronal Multicapa	14
Funciones de procesamiento de entradas y salidas	16
Redes neuronales en sistemas adaptativos	16
Red Neuronal Perceptron	17
Procesos de aprendizaje de Perceptron	18
Ejercicios Matlab® y Analíticos.....	19
Perceptron de múltiples neuronas.....	25
Limitaciones del Perceptron.....	26
Red neuronal Adaline y Madaline	28
Procesos de aprendizaje Adaline	29
Filtros adaptativos de redes neuronales.....	34
Funciones adaptativas.....	34
Referencias.....	35
Capítulo II - Algoritmos de entrenamiento de redes: propagación hacia atrás y otros algoritmos... ..	36
Introducción	36
Método del Gradiente descendente para red neuronal de 2 capas	37
Método de gradiente descendente para múltiples capas	42
Mejoras en el rendimiento del algoritmo de propagación hacia atrás.....	44
Método del Gradiente descendente con velocidad de aprendizaje variable	44
Método del gradiente descendente con momentos	45
Método de Newton.....	45
Algoritmo de Levenberg Marquardt	46
Ejercicios Matlab®, Python y Analíticos	48

Referencias.....	63
Capítulo III - Redes de funciones de base radial (RBF).....	64
Introducción	64
Estructura de redes RBF.....	65
Funciones RBF	66
Análisis locales y globales.....	68
Procedimiento de aprendizaje de las redes RBF.....	68
Mínimos cuadrados lineales en RBF (linear Least squares: LLS).....	69
Mínimos cuadrados ortogonales	72
Agrupamiento (Clustering) en RBF.....	72
Optimizaciones no lineales.....	74
Ejercicios de redes RBF.....	74
Referencias.....	83
Capítulo IV - Reconocimiento estadístico de patrones, aprendizaje automático y redes neuronales	84
Reconocimiento de patrones con redes neuronales	84
Descripción del reconocimiento de patrones	84
Ajuste de datos o Regresión.....	86
Reconocimiento de Patrones - Clasificación.....	86
Agrupamiento (clustering)	86
Datos de entrenamiento, de validación y de testeo	88
Ejercicios.....	88
Reconocimiento estadístico de patrones.....	94
Técnicas de reconocimiento de patrones	95
Similitud	97
Clúster de particiones	98
Clúster jerárquico	98
Análisis de Componentes Principales (PCA).....	99
Clasificadores	102
Medición y validación cruzada	102
Análisis de funciones discriminantes (DFA)	103
Análisis discriminantes lineales mediante método de Fisher	104

Método K-NN	105
Análisis de datos y sensores.....	106
Estimación de la función de densidad de probabilidad	111
Preprocesamiento, normalización y extracción de características o parámetros	116
Exploración de datos mediante métodos gráficos.....	118
Aprendizaje y generalización	119
Evaluación de la generalización: conjuntos de entrenamiento, prueba y validación.....	119
Entrenamiento y generalización	120
Métodos de validación.....	120
Análisis de sensibilidad de la red.....	121
Validación cruzada	121
Matriz de Confusión	123
Referencias.....	127
Capítulo V - Redes Neuronales con Mapas Autoorganizados (SOM).....	129
Introducción	129
Ejercicios.....	131
Referencias.....	136
Capítulo VI - Redes Neuronales Dinámicas	137
Introducción	137
Estructuras de Redes estáticas, dinámicas prealimentadas y dinámicas recurrentes.....	138
Aprendizaje en redes dinámicas	138
Predicción e Identificación con redes neuronales dinámicas para aplicaciones de series temporales no lineales.....	141
Redes neuronales recurrentes (RNN)	141
Ventajas de las RNN. Procesamiento natural de lenguaje (PNL)	143
Clasificación de RNN.....	143
Redes recurrentes LSTM y GRU.....	144
Ejercicios de Redes neuronales dinámicas.....	146
Referencias.....	176
Capítulo VII –Redes Neuronales Convolucionales.....	177
Introducción a las Redes Neuronales Convolucionales (CNN)	177
Herramientas y librerías avanzadas para redes CNN	180

Ejercicios.....	181
Referencias.....	207
Capítulo VIII – Inteligencia Artificial en Dispositivos Móviles y de IoT	208
Introducción	208
Tutoriales y ejemplos de Tensorflow para descargar	209
Detalles de programas de clasificación de imágenes con Tensorflow.....	212
Modelos de redes neuronales de aprendizaje profundo en dispositivos móviles.....	213
Creación de un modelo propio en TensorFlow Lite	214
Descarga de ejemplos varios en TensorFlow Lite	214
Tutoriales y ejemplos de Mit App Inventor para descargar.....	215
Algunas aplicaciones destacadas de MIT	215
Referencias.....	216
Capítulo IX – Visión Artificial	217
Introducción	217
Detección de bordes con OpenCV mediante función Canny	218
Detector de objetos con SimpleBlobDetector() de OpenCV.....	218
Detección de objetos con clasificadores en cascada Haar.....	219
Detección de esquinas con detector de Harris (Harris corner Detection).....	220
Shi-Tomasi con función goodFeaturesToTrack	221
Comparación con patrones (Template matching)	222
Ejercicios.....	223
Referencias.....	235
Capítulo X - Aplicaciones con métodos combinados de Inteligencia Artificial	236
A) Clasificación de vinos. Ejemplo de combinación de Análisis Multivariado de datos, clasificadores y redes neuronales con software Python.....	236
B) Ejercicio de Algoritmos combinados con software Orange y conjunto de datos Iris	241
C) Algoritmos combinados para aplicaciones de Espectroscopía de Plasma Inducida por Láser .243	
Introducción a mediciones LIBS	243
Algoritmos de reconocimiento de patrones aplicados a LIBS.....	245
Ejemplo de Mediciones de suelos con técnicas LIBS	246
Resultados de Mediciones LIBS.....	248
D) Comparación de redes neuronales y diferentes técnicas de inteligencia artificial con Python y software RapidMiner	249

Referencias.....	253
Capítulo XI -Aplicaciones. Procesado de señales interferométricas con redes neuronales: Estimando frecuencias	256
Introducción	256
Interferometría de baja coherencia	257
Caso de estudio: Perfilometría/topografía	263
Estimación de frecuencia	266
Consideraciones generales.....	267
Estimación paramétrica de frecuencias con redes neuronales	269
Estimación de frecuencias a partir de la señal completa.....	273
Ganando profundidad para estimar frecuencias de señales completas.....	282
Conclusiones	291
Referencias.....	292
Capítulo XII - Redes Neuronales aplicadas a la detección de picos de glucosa después de las comidas en series temporales de pacientes diabéticos tipo II.....	295
Analizando los datos	295
Entrenando la red.....	299
Probando la red.....	301
Referencias.....	304
Anexo I – Lista de ejercicios	305
Epílogo.....	312
Contenido	313

Juan Vorobioff
Santiago Cerrotta
Nicolas Eneas Morel
Ariel Amadio

Inteligencia Artificial y Redes Neuronales

Fundamentos, Ejercicios y Aplicaciones con Python y Matlab



Este libro se basa en la experiencia profesional de los autores en el área de Inteligencia Artificial (IA) aplicado al análisis y procesamiento de datos, señales e imágenes.

La IA contiene distintas disciplinas como aprendizaje automático, aprendizaje profundo, minería de datos, entre otras. Analizamos principalmente las redes neuronales.

Sin embargo, también mostramos algunos métodos de reconocimiento estadístico de patrones y métodos de visión artificial, ya que presentan algunas ventajas (y desventajas) respecto de las redes neuronales. Dentro de las redes neuronales presentamos una introducción a las redes básicas, a las redes convolucionales de aprendizaje profundo y a las redes dinámicas.

Para cada tema se presenta un marco teórico, desarrollos matemáticos, ejercicios analíticos, ejemplos de aplicaciones y ejercicios en Matlab®, Python y otros entornos. El libro contiene los códigos QR para descargar los programas. Se espera que el contenido del libro le sirva al lector para comprender diferentes técnicas de IA, así también para desarrollar y comparar distintos algoritmos con aplicaciones científico tecnológicas y/o comerciales.

ISBN 978-987-4998-82-8



9 789874 998828