Universidad Nacional Autónoma de México

Facultad de Ciencias

PREDICCIÓN DE ALARMAS DE FALLAS EN REDES DE TELECOMUNICACIONES CON REDES NEURONALES ARTIFICIALES

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

MATEMÁTICO

P R E S E N T A :

ALEJANDRO SALCIDO PARADA

TUTOR

M. en I.A. José Luis Jiménez Andrade

Ciudad Universitaria, Cd. Mx., 2020

VERSIDAD NACIONAL
AVFNOMA DE
MEXICO

FACULTAD DE CIENCIAS
Secretaría General
División de Estudios Profesionales

Votos Aprobatorios

LIC. IVONNE RAMÍREZ WENCE
Directora General
Dirección General de Administración Escolar
Presente

Por este medio hacemos de su conocimiento que hemos revisado el trabajo escrito titulado:

PREDICCIÓN DE ALARMAS DE FALLAS EN REDES DE TELECOMUNICACIONES CON REDES NEURONALES ARTIFICIALES

realizado por **Alejandro Salcido Parada** con número de cuenta **410086036** quien ha decidido titularse mediante la opción de **tesis** en la licenciatura en **Matemáticas**. Dicho trabajo cuenta con nuestro voto aprobatorio.

Propietaria	Dra. Ursula Xiomara Iturrarán Viveros	Ursula Iturrarán
Propietario	Dr. Humberto Andrés Carrillo Calvet	HCarrillo
Propietario Tutor	M. en I.A. José Luis Jiménez Andrade	
Suplente	Dr. Alessio Franci	AF
Suplente	Mat. Salvador López Mendoza	

Atentamente
"POR MI RAZA HABLARÁ EL ESPÍRITU"
CIUDAD UNIVERSITARIA, CD. MX., A 22 DE OCTUBRE DE 2019

JEFE DE LA DIVISIÓN DE ESTUDIOS PROFESIONALES

ACT. MAURICIO AGUILAR GONZÁLEZ

Señor sinodal: antes de firmar este documento, solicite al estudiante que le muestre la versión digital de su trabajo y verifique que la misma incluya todas las observaciones y correcciones que usted hizo sobre el mismo.

A mi familia pero principalmente a mi madre que siempre me ha apoyado.

Índice general

Agradecimientos

Para elaborar esta tesis muchas personas me ayudaron de una u otra manera: entender conceptos básicos, aprender a usar herramientas de software necesarias y un sin fin de aspectos que sin ellos me hubiese sido imposible escribir este texto. La lista es muy larga y es más prudente un agradecimiento general . Sin embargo, agradezco especialmente la paciencia y dedicación de mi asesor José Luis Jiménez Andrade al acompañarme en mis primeros pasos en el mundo de las redes neuronales e incorporarme al cubículo de Dinámica No Lineal de la Facultad De Ciencias de la UNAM, en donde he pasado ratos muy agradables y provechosos en el ámbito académico en compañía de sus integrantes. También a mis sinodales, quienes se tomaron la titánica, y ciertamente en ocasiones molesta, labor de leer y hacer no pocas correcciones a este texto.

Prólogo

Las redes de telecomunicaciones han entrado en una era de cambios fundamentales (véase el prefacio de [17]). Solemos percatarnos de su trascendencia cuando ocurren cierta clase de eventos que alteran su buen funcionamiento . Considérese situaciones relativamente comunes, como un sismo por ejemplo, en las cuales estas redes pueden fallar y causar un caos en nuestra vida diaria. Pero las fallas en las redes de telecomunicaciones no están asociadas únicamente a fenómenos externos, existe otra clase de eventos que pueden provocar caídas en los servicios.

En las reuniones que el autor de esta tesis ha tenido con personas encargadas de la gestión de redes de telecomunicaciones, se ha discutido la necesidad de anticiparnos a la presencia de eventos que pueden derivar en fallas. Si bien hay literatura al respecto (véase por ejemplo [18]), se comentó en esas reuniones que las empresas proveedoras de servicios de telecomunicaciones en México no cuentan con herramientas de software eficaces para tal tarea. Esto, aunado al auge de la aplicación del aprendizaje profundo o *Deep Learning* en diversos problemas (un vistazo a la introducción de [13] ofrece un panorama), motivó a proponer el uso de las redes neuronales para el problema de la anticipación de eventos que deriven en fallas.

El presente trabajo aborda el problema de predicción de alarmas de falla en redes de telecomunicaciones usando redes neuronales artificiales. En particular se hará uso de la red neuronal conocida como *Perceptrón multicapa* con el objetivo de aprovechar los registros de alarmas de fallas almacenados en las bases de datos de las compañías de telecomunicaciones.

Los resultados de este trabajo demuestran el gran valor que podría aportar un sistema basado en inteligencia artificial, específicamente, basado en redes neuronales artificiales, para mejorar la calidad de los servicios que ofrecen las compañías de telecomunicaciones. Los experimentos realizados en esta tesis con historiales de alarmas reales demuestran que la red neuronal puede predecir hasta un 84 % de las fallas contenidas en esa base.

El trabajo se organiza de la siguiente manera: en el primer capítulo se presentan los conceptos mínimos necesarios sobre redes de telecomunicaciones y gestión de fallas. Es el tema de gestión de fallas el que motiva abordar posibles herramientas

que nos ayuden a predecirlas. El segundo capítulo presenta un enfoque bastante socorrido en la predicción de alarmas en redes de telecomunicaciones llamado minería de patrones en secuencias de eventos. En este capítulo se introducen algunos conceptos relativos a este enfoque que nos serán de ayuda para plantear el problema cuando usemos redes neuronales artificiales. El tercer capítulo aborda el tema central de la tesis: las redes neuronales artificiales. El objetivo es describir la matemática detrás de la red neuronal conocida perceptrómulticapa y el algoritmo insignia para su entrenamiento: el algoritmo de retropropagación. Se mencionan algunas de las técnicas para mejorar el aprendizaje de una red neuronal utilizando el algoritmo de retropropagación. El cuarto capítulo propone maneras de introducir las alarmas para que puedan ser entendidas por la red neuronal con estructura de perceptrón multicapa. El quinto capítulo describe algunos experimentos con una base de datos de alarmas reales y datos sintéticos. Los experimentos consisten en ajustar los parámetros en la red neuronal para obtener resultados satisfactorios. El sexto capítulo muestra las conclusiones de la tesis. Por último, los anexos A y B dan una breve introducción a las herramientas computacionales utilizadas en esta tesis.

Algunos de los capítulos son prescindibles si ya se conocen las herramientas o si se desea conocer directamente los resultados de esta tesis.

Capítulo 1

Gestión de fallas en redes de telecomunicaciones

Esta sección aborda de manera somera el tema de las fallas en las redes de telecomunicaciones. **La información aquí contenida puede consultarse con más detalle en [16] y [17]**.

1.1. Redes de telecomunicaciones

Para efectos de esta tesis, una red de telecomunicaciones la podemos entender como un conjunto de medios y tecnologías que proveen un servicio: la transferencia de información entre usuarios localizados en diferentes posiciones geográficas . Las redes de telecomunicaciones son diseñadas para satisfacer ciertas necesidades. Algunos de los servicios que satisfacen son: correo electrónico, servicio telefónico, servicio de telefonía celular, internet, radio, televisión, entre otros [17].

1.2. Gestión de redes

Para mantener la calidad en el servicio es necesario supervisar que la red de telecomunicaciones realice sus tareas. La gestión de redes incluye el despliegue, integración y coordinación del hardware, software y los elementos humanos para monitorizar, probar, sondear, configurar, analizar, evaluar y controlar los recursos de la red, para satisfacer los requerimientos de tiempo real, en el desempeño operacional y la calidad de servicio a un precio razonable [16]. Gracias a la gestión de la red evitaremos que ésta funcione incorrectamente degradando sus prestaciones. Este tipo de gestión se realiza mediante un sistema de gestión de la red [16].

1.3. La gestión de las fallas

Dentro de la red viajan mensajes que proporcionan reportes del estado de los elementos que conforman la red. Estos eventos son capturados por un sistema de gestión de fallas [16].

En la figura 1.1 se presenta el aspecto que tiene una alarma de falla real.

```
Log Record ID:                  359430736
Event Time:                     2013-10-08    16:34:00

Event Type:                     O2
Object of Reference:            SubNetwork=ONRM_ROOT_MO,SubNetwork=AXE-R9,ManagedElement=CUATR1,BssFunction=BSS_ManagedFunction,BtsSiteMgr=MX3487
Object Class:                   8
Backup Object:
Backup Status:
Trend Indication:
Perceived Severity:             Minor

Probable Cause:                 Different causes possible for same message

Specific Problem:               RADIO X-CEIVER ADMINISTRATION MANAGED OBJECT FAULT

Problem Text:
*** ALARM 650 A3/APT "CUATR1 UM070005" 131008   1634
RADIO X-CEIVER ADMINISTRATION
MANAGED OBJECT FAULT

MO                               RSITE          ALARM SLOGAN
RXOCF-437                        MX3487         BTS INTERNAL
END


Problem Data:


Proposed Repair Action:
No value
Correlated ID:
Acknowledge Time:
Acknowledged by:

Comments:

Alarm Class:                    A3
Alarm Category:                 No value
Alarm ID:                       359430736
Alarm Number:                   650
Attendance Indication:
Record Type:                    Synchronisation Alarm
Object Type:                    40958044
Logging Time:                   2013-10-08    22:02:27
Cease Record Type:              Alarm
Cease Time:                     2013-10-09    11:46:00
FMX Generated:

Additional attributes
```

Figura 1.1: Información de una alarma real.

1.4. Predicción de fallas

El sistema que gestiona la red puede exhibir un comportamiento reactivo, esto es, toma una decisión al recibir cierta falla. Sin embargo, la complejidad de las redes de telecomunicaciones y la posibilidad de que se propague alguna falla en la red que pueda ocasionar la denegación del servicio a los usuarios finales, hace deseable que la red de telecomunicaciones exhiba un comportamiento preventivo [16].

Para este propósito diversas técnicas de predicción de fallas y técnicas de corrección de fallas han sido propuestas. La mayoría de éstas técnicas están basadas en el análisis histórico del registro de alarmas. Las técnicas comúnmente utilizadas

son: sistemas basados en reglas, cadenas de Markov, redes bayesianas, lógica difusa y redes neuronales [15].

Capítulo 2

Patrones en una secuencia de alarmas

2.1. Introducción

En las redes de telecomunicaciones se mantiene un registro de las alarmas ocasionadas por fallas en la red, las cuales son recabadas por un nodo central. La hipótesis con la que se trabaja en esta tesis es que el registro de alarmas puede contener patrones a partir de los cuales se podrían hacer predicciones, con cierto grado de certeza, acerca de la ocurrencia de una o varias fallas.

2.2. La información temporal y su representación

La información temporal se refiere a asociar una etiqueta temporal a contenedores de información de un sistema. Esta asociación tiene como objetivo describir los cambios de estado del sistema o su evolución a lo largo del tiempo.[18]

La información temporal es dividida en dos clases: información de eventos y series de tiempo. En la información de eventos, los valores que toma la variable a predecir son discretos y cualitativos; en series de tiempo la variable toma valores en un continuo. Es común hacer uso de la minería de los datos de eventos en redes de telecomunicaciones. [18]

Dentro de la modelación de eventos existen diferentes definiciones para patrones de eventos -en virtud de las diversas maneras de representar secuencias de eventos -, técnicas y escenarios de aplicación. Una exposición más detallada de estos patrones y técnicas se puede encontrar en [18] y en [12]. En este trabajo usaremos el enfoque de patrones de eventos abordado por Hannu Toivonen y A. Inkeri Verkamo en [19].

2.2.1. Patrones en secuencia de eventos

Las alarmas que se generan en una red de telecomunicaciones poseen información diversa (vease la figura 1.1). Una manera de encontrar patrones en las alarmas recibidas es asignar una categoría a cada alarma (como en [18]). En esta tesis asumiremos que tal categorización fue hecha con anterioridad.

Al registro de alarmas o eventos en el tiempo lo llamaremos *secuencia de eventos*. Esta secuencia describe el comportamiento de una red de telecomunicaciones pero puede también describir los actos delictivos de una persona, un organismo biológico, transacciones financieras, entre otras.

2.3. Eventos y alarmas

Ahondaremos en algunos conceptos y métodos propuestos por Heikki Mannila, Hannu Toivonen y A. Inkeri Verkamo [19] para descubrir patrones en una secuencia de eventos. Registramos un evento en la secuencia cuando conocemos la alarma que se generó y su respectivo tiempo de arribo. Hacemos la convención de que el tiempo es discreto.

Los eventos (alarmas) serán etiquetados para facilitar su manejo. Podemos asociar un tipo de alarma a una letra del abecedario, por ejemplo al evento de un equipo mal configurado podemos asociarle la letra “B”.

Un ejemplo de una secuencia de eventos podría ser:

{(D,10), (C, 20), (A, 30), (B, 40), (D, 50), (B,60), (A,80), (B,90),(D,100), (A,110), (B,1209), (D,130),(A,140)}

El evento (A,7) se interpreta como “se registró una falla de un componente, identificada con la etiqueta A, en el tiempo 7”. Toda alarma tiene que ser de algún tipo y por ello tiene que tener alguna etiqueta. El conjunto de tipos de alarmas además tiene que ser finito.

La secuencia de eventos anterior puede ser modelada con una línea de tiempo como se muestra en la figura 2.1.

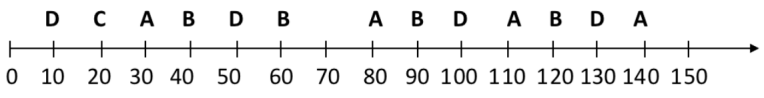


Figura 2.1: Secuencia de tiempo

El tiempo está dado en ciertas unidades (segundos, minutos, días, meses, años, etc) y se considera que cada evento se recibe en un tiempo entero.

En el ejemplo, a simple vista podemos apreciar un patrón: a la ocurrencia del evento (alarma) B, casi siempre le precede la ocurrencia del evento A (exactamente en ese orden). La "distancia" entre la ocurrencia de ambos eventos tampoco parece ser considerable. Esta noción de cercanía de eventos y de frecuencia de ocurrencia es lo que reconocemos como un patrón o *episodio*. Definiremos con más formalidad estas nociones a continuación.

2.4. Episodios

Diremos que un patrón (episodio de aquí en adelante) es un conjunto de eventos que ocurren "muy cerca" en la secuencia de eventos o registro de las alarmas. Estos eventos tienen asignados sus respectivas etiquetas. En el ejemplo mostrado en la imagen 2.1, podemos percatarnos de que no suele haber más de 50 unidades de tiempo a partir de la ocurrencia de A y la posterior ocurrencia de algún evento B (en ese orden). En ese mismo ejemplo es posible que entre la ocurrencia de A y B se "entrometa" otro evento cualquiera, por ejemplo C, pero para decir que se cumple el episodio "A precede a B" a nosotros solo nos importará que:

1. Suceda primero (algún) A.
2. Suceda después (algún) B.
3. Ambos sucedan dentro de un intervalo de tiempo definido, digamos 50 unidades de tiempo.

2.5. Ventana

La tercera condición origina el concepto de *ventana*, un intervalo que coincide con ser un "extracto" o pedazo de la secuencia original de la cual podemos especificar el tiempo en que empieza y el tiempo en que termina. La diferencia entre ambos tiempos es la *longitud o tamaño de la ventana*.

Podemos fijar la longitud de la ventana y hacer que comience en cualquier tiempo de la secuencia de eventos original. Más aún, es posible que el tiempo de inicio de la ventana sea anterior al tiempo de inicio de la secuencia original. Esto es posible siempre y cuando el último tiempo de la ventana se encuentre dentro de la secuencia original. Además, nada nos impide considerar a la secuencia de eventos

completa como una ventana (una de las de tamaño más grande que puede haber en la secuencia).

A manera de ejemplo, en la figura 2.2 se muestra una secuencia de eventos que empieza en 0 y termina en 90. Además, se muestran todas las ventanas de tamaño 40 (es decir, la diferencia entre el tiempo final y el tiempo inicial es de cuarenta unidades) que se pueden formar en esa secuencia de eventos. Para generarlas, desplazamos la ventana cada 10 unidades de tiempo, es decir, dos ventanas consecutivas difieren en su punto inicial por 10 unidades de tiempo. A esto le llamamos *paso de ventana*.

Pediremos que el punto final de cualquier ventana sea mayor que el punto inicial de la secuencia de eventos. De igual manera se pide que el punto inicial de cualquier ventana sea menor que el punto final de la secuencia original. Por eso no podemos colocar una ventana cuyo punto final sea el 0, el cual corresponde al principio de la secuencia.

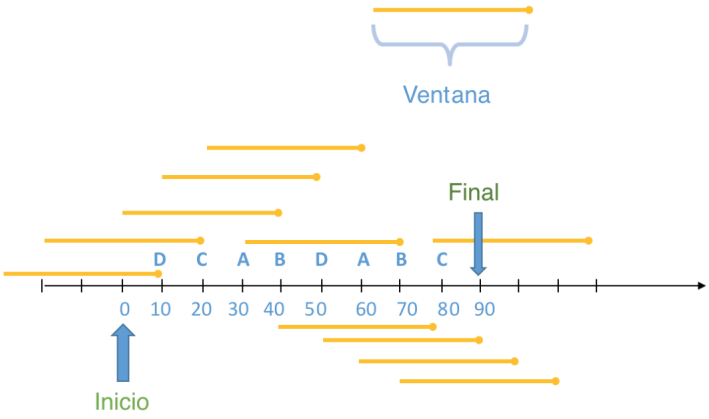


Figura 2.2: Ventanas en una secuencia de alarmas

Nuestro objetivo será predecir, a partir de los eventos que se presentan en una ventana, la presencia de una alarma que suceda después del último elemento de la ventana.

Capítulo 3

Redes neuronales Artificiales

3.1. Introducción

Las redes neuronales biológicas proporcionan un marco de referencia para desarrollar herramientas matemáticas como la red neuronal artificial que se explicará en este capítulo. Sin embargo, no se pretende imitar o simular de manera fidedigna una red neuronal biológica; ora por la dificultad que ello plantea, ora porque nuestro interés está puesto en aplicaciones que no son realizables de manera común por nuestra mente (como predecir a partir de una base de datos enorme alguna tendencia en cuestión de segundos). De tal suerte que el presente trabajo se toma cierta libertades en favor de una postura usualmente pragmática. Una breve discusión sobre las diferencias de una red neuronal artificial y biológica puede consultarse en el primer capítulo de [10].

En esta tesis se usará la expresión "modelo matemático" como una selección adecuada de ciertas características de un objeto de estudio que nos permiten describir su comportamiento (véase la introducción de [9]). En [10] se establece que las redes neuronales biológicas son un enfoque socorrido dentro del campo de la inteligencia artificial. No obstante, se recurre a simplificaciones de las características de las redes neuronales biológicas para enfocarnos en aspectos que se consideran útiles, o bien se intenta imitar alguna de las habilidades que nos proporciona la inteligencia por separado mediante una red neuronal artificial (como lo podría ser la detección de imágenes). En el modelado se hace uso de diversas herramientas matemáticas (teoría de la probabilidad, estadística, análisis, geometría, etc).

3.2. Un modelo simple de neurona

La idea simplificada que nos proporciona la biología sobre una neurona nos lleva a un esquema de grafo dirigido (gráfica dirigida) como se muestra en la figura 3.1.

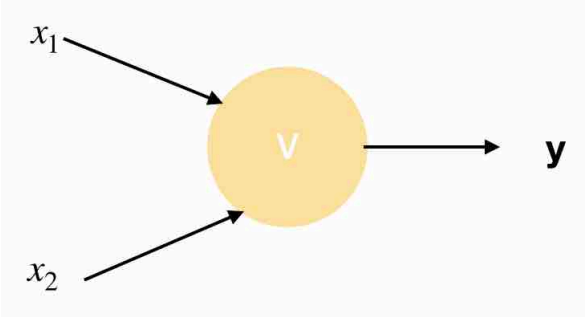


Figura 3.1: Esquema de una neurona.

En el esquema tenemos un nodo V , que representa a la neurona, el cual recibe "señales" o estímulos x_1, x_2 a través de los enlaces de conexión y procesa esas señales para arrojar una respuesta y. Cada señal fluye en el sentido que las flechas indican. En principio la naturaleza de las señales podría ser de diversa índole, sin embargo nosotros las codificaremos como números reales. Dentro del nodo se realizará un procedimiento que nos arrojará un valor y como respuesta a las señales o estímulos x_1 y x_2, a semejanza de una neurona real. La respuesta y puede ser también de diversa índole, nosotros asumiremos que y es simplemente un número.

Utilizaremos vectores para modelar la entrada de información, estímulos o señales a la neurona. Tomaremos pues $X = (x_1, x_2) \epsilon \mathbb{R}^2$ como entradas de nuestra neurona V y su salida simplemente como $y \epsilon \mathbb{R}$, donde $\mathbb{R}$ denota al conjunto de los números reales y $\mathbb{R}^2$ al conjunto de pares ordenados donde cada entrada es un número real. Si bien el vector X pertenece a $\mathbb{R}^2$, en la práctica las entradas (*inputs*) podrían ser binarias.

Nuestra neurona V tendrá como procedimiento asociado retomar la información del vector $X = (x_1, x_2)$ y ponderar (asignarles algún peso según la importancia que tengan en caso de ser necesario) a cada entrada. Lo anterior nos acerca al mecanismo básico de plasticidad sináptica. [1] Dicho mecanismo fue introducido por Donald Hebb y nos será útil cuando le permitamos a nuestra neurona "aprender" a catalogar de mejor manera las entradas.

En lenguaje matemático, la labor que realizará el nodo es la operación $w_1 * x_1 + w_2 * x_2 = X \cdot W = V(X)$, donde el vector $W = (w_1, w_2)$ nos proporciona la

[1] De acuerdo a la teoría hebbiana, dicho mecanismo sugiere que el valor de una conexión sináptica de una neurona se incrementa si las neuronas de ambos lados de dicha sinapsis se activan repetidas veces de forma simultánea. [1]

ponderación de los estímulos que recibimos. En sintonía con lo que sucede en una neurona real, siempre que se supere un cierto umbral, que modelaremos como un número constante C, se tendrá una respuesta y. De momento haremos que $y \in \{0, 1\}$.

Así pues, es posible modelar el procedimiento del nodo V como una función lineal de la siguiente manera:

$$V(X) : \mathbb{R}^2 \to \mathbb{R}$$

Cuya regla de correspondencia es:

$$V(X) = V((x_1, x_2)) = \begin{cases} 1 & w_1 * x_1 + w_2 * x_2 > c \\ 0 & w_1 * x_1 + w_2 * x_2 \leq c \end{cases} \tag{3.1}$$

A la función $V(X)$ la llamaremos *función de activación de la neurona o nodo.* La función de activación define la salida de un nodo dada una entrada o un conjunto de entradas (la entrada a la que que hemos denotado con el vector $X = (x_1, x_2) \in \mathbb{R}^2$).

Hay que recordar que el dominio de la función $V(X)$ podría no ser todo $\mathbb{R}^2$ sino un subconjunto $A \subset \mathbb{R}^2$.

Si consideramos w_1 y w_2 como constantes la ecuación $w_1 * x_1 + w_2 * x_2 = c$ nos representa una recta en el plano cartesiano donde la variable x_1 es independiente y x_2 es la variable dependiente. Para hacerla más reconocible despejamos a x_2 y obtenemos que: $x_2 = \frac{c}{w_2} - \frac{w_1}{w_2} * x_1$. La ecuación anterior nos dibuja una recta en el plano cartesiano de pendiente $-\frac{w_1}{w_2}$ y con intersección en el eje de las abscisas $\frac{c}{w_2}$.

3.3. ¿Qué puede hacer una sola neurona con ese funcionamiento tan simple?

Con una neurona podemos clasificar nuestras entradas o *inputs*. Cada entrada que recibimos, codificada como un vector X en el plano cartesiano, queda "clasificada" como 1 o 0 dependiendo de lo que la función $V(X) = y$ nos arroje. Tenemos pues dos categorías en las que podemos clasificar cada entrada que recibimos. Cualquier vector de entradas $X \epsilon \mathbb{R}^2$ que cumpla que $X \cdot W > c$ será clasificado como 1. De lo contrario el vector X será etiquetado como 0.

Podríamos dar un nombre especial a esos vectores $X \epsilon \mathbb{R}^2$ tales que al evaluarlos en la función $V(X)$ nos de 0 o bien 1. Lo anterior es la definición de las imágenes

inversas de los conjuntos $\{0\}$ y $\{1\}$ bajo la función $V(X)$. Formalmente se define:

$$V^{-1}[\{0\}] = \{X \in \mathbb{R}^2 \mid V(X) = 0\} \tag{3.2}$$

$$V^{-1}[\{1\}] = \{X \epsilon \mathbb{R}^2 \mid V(X) = 1\} \tag{3.3}$$

Llamemos a $V^{-1}[\{0\}]$ como la región R_0 y a $V^{-1}[\{1\}]$ como la región R_1 en virtud de las dos categorías en las que dividimos las entradas. Tenemos una interpretación geométrica y una interpretación analítica.

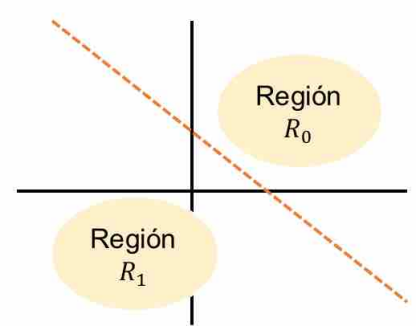


Figura 3.2: Regiones inducidas por las etiquetas de las entradas.

Gracias a esta perspectiva podemos clasificar realizando simples cálculos. Con base en los estímulos que recibe nuestra neurona podemos decidir si debemos dar vuelta a la izquierda o a la derecha en un automóvil (si asignamos el significado de la salida 0 de la neurona como "vuelta a la izquierda" y la salida 1 como "vuelta a la derecha").

Nuestra intención al usar redes neuronales es poder clasificar cierta información de entrada en alguna categoría. Aplicaciones como el reconocimiento de imágenes serán posibles con algún grado de certeza.

Dado que la recta que sirve de "criterio" para clasificar depende del vector W, nuestra clasificación de los *inputs* podría ser errónea. Al ser nuestro modelo de neurona una réplica de una neurona real, podemos mejorar la labor de clasificación que realiza. Nuestra neurona artificial va a intentar mejorar su labor de clasificación a partir de nuestra supervisión (le diremos cuándo hizo bien su trabajo y cuándo no) . Eso nos llevaría a que, además de *clasificar*, la neurona artificial también puede *aprender*.

Para mejorar nuestra labor de clasificación es posible "empujar" la recta que nos separa las regiones R_1 y R_2 (en dirección horizontal o vertical). Notemos que el vector W , que hemos modelado como el peso de nuestras conexiones sinápticas, es posible modificarlo con cierto grado de libertad en aras de clasificar adecuadamente nuestros *inputs*.

El efecto de modificar el vector W se observa mejor si recordamos el siguiente resultado que nos proporciona el álgebra lineal o la geometría:

$$X \cdot W = |X| * |W| * cos(\theta)$$

- $|X|$: La magnitud de la longitud del vector. Formalmente representa la norma euclidiana usual en $\mathbb{R}^n$ ($\mathbb{R}^2$ en nuestro caso) cuyo cálculo se limita a hacer la operación $\sqrt{X \cdot X}$.
- θ: El ángulo que hay, en el plano cartesiano, entre los vectores X y W . Como convención se suele tomar en el sentido contrario al movimiento de las manecillas del reloj.

En general podemos definir un producto punto entre dos vectores que no necesariamente coincide con el producto punto usual (el definido unas líneas arriba) y también podemos definir normas que no necesariamente coinciden con la norma que hemos utilizado.

3.4. Conjuntos linealmente separables

Cuando dos conjuntos de entradas, digamos $A, B \subset \mathbb{R}^2$, los podemos separar por una recta parametrizada por el vector $W \epsilon \mathbb{R}^2$, decimos que esos conjuntos son linealmente separables. No puede suceder que "arriba" de la recta tengamos un punto que pertenezca a ambos conjuntos A y B por ejemplo.

El procedimiento definido por la neurona $V(X)$ nos permite clasificar cada vector de entradas según se encuentren por debajo o arriba de la recta que define el vector de conexiones sinápticas $W \epsilon \mathbb{R}^2$.

Otra interpretación del procedimiento de nuestra neurona V es que intenta aproximarse a una función que existe idealmente, la cual podemos definir como $I(X) : \mathbb{R}^2 \to \mathbb{R}$, que al evaluar cada vector de entradas $X \epsilon \mathbb{R}^2$ nos proporciona su "verdadero" valor. Cabe resaltar que de ésta última función solo conocemos algunos de sus valores.

3.5. Limitaciones

La neurona así construida únicamente distingue dos categorías; decide si una entrada pertenece o no a una categoría dependiendo si el vector X se encuentra "arriba" de la recta o "debajo" de la recta. Podemos hacer una nueva selección de

$W \epsilon \mathbb{R}^2$ pero solo modificaría la posición de nuestra recta que nos sirve de "criterio" para clasificar nuestros entradas.

Interpretando la labor de una neurona como una aproximación de cualquier función, no podemos hacer que la función $V(X)$ (la función de la neurona) aproxime cualquier función arbitraria. O dicho con otras palabras, no podemos dibujar una recta en el plano tal que cada entrada $X \epsilon \mathbb{R}^2$ este correctamente clasificada en la región que debiera estar.

Un ejemplo sencillo ilustra esta situación. La función lógica conocida como disyunción exclusiva tiene como definición:

$$XOR : \mathbb{N}^2 \to \mathbb{N}$$

Con regla de correspondencia:

$$\begin{aligned} XOR(0,0) &= 0 \\ XOR(0,1) &= 1 \\ XOR(1,0) &= 1 \\ XOR(1,1) &= 0 \end{aligned}$$

Siendo 0 el estado lógico conocido como falso y 1 el estado lógico conocido como verdadero.

En este ejemplo hay cuatro vectores de entrada y son:

$$\begin{aligned} X_1 &= (0,0) \\ X_2 &= (1,0) \\ X_3 &= (0,1) \\ X_4 &= (1,1) \end{aligned}$$

La gráfica de esta función se muestra en la imagen 3.3:

Siendo A el conjunto de los puntos blancos y B el conjunto de los puntos en negro, no es posible hallar una recta que nos separe estos conjuntos como quisiéramos pues "debajo" o "arriba" de cualquier recta propuesta habría puntos tanto de A como de B .

Figura 3.3: Ejemplo de no separabilidad lineal.

3.6. El perceptrón

A pesar de la limitación que vimos anteriormente, quisiéramos obtener el mayor provecho posible a este modelo simple de neurona. Dicho modelo es conocido con el nombre de *perceptrón* y nos será útil cuando, de antemano, sepamos que los datos, representados en $\mathbb{R}^n$, son linealmente separables. Aunque en toda la discusión hemos trabajado en $\mathbb{R}^2$ la generalización a $\mathbb{R}^n$ es muy sencilla.

Recordemos que tenemos total libertad de cambiar el vector $W \epsilon \mathbb{R}^n$ que representa los pesos sinápticos. De acuerdo a [1], el algoritmo utilizado para ajustar éste vector surgió como un procedimiento de aprendizaje de Rosenblatt (1958). Antes de comentar el algoritmo, haremos algunas adecuaciones al modelo que seguimos. Dicho esquema podemos visualizarlo mejor en la imagen 3.4.

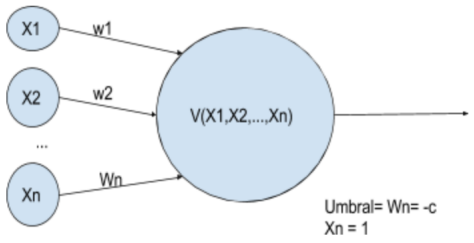


Figura 3.4: Esquema de un perceptrón

La gráfica no ha cambiado mucho salvo que hemos considerado una entrada fija y una entrada del vector $W \epsilon \mathbb{R}^n$ como fija (esa entrada es el valor umbral). La neurona hace una labor ligeramente distinta.

$$V(X) = \begin{cases} 1 & W \cdot X > 0 \\ 0 & W \cdot X \leq 0 \end{cases} \tag{3.4}$$

Nótese que la anterior función nos proporciona el mismo resultado pues:

$$X \cdot W = w_1 * x_1 + w_2 * x_{2..} ... + w_n * x_n.$$

Pero:

$$w_n = -c \text{ y } x_n = 1$$

Por lo que:

$$X \cdot W = w_1 * x_1 + w_2 * x_{2..} ... + (-c) * (1)$$

Si igualamos a cero y despejamos tenemos que:

$$w_1 * x_1 + w_2 * x_{2..} ... + w_{n-1} * x_{n-1} = c$$

La anterior ecuación nos proporciona la ecuación de un hiperplano cuya dimensión es n-1 (la última entrada tanto del vector W como del vector X son fijas, valen -C y 1 respectivamente). Las entradas originales están dadas por las entradas 1, 2,3,..., n-1 del vector $W \epsilon \mathbb{R}^n$.

En el caso de la función XOR, nuestras entradas a la red neuronal serían de la forma:

1. V (0,0,1)=0
2. V (0,1,1)=1
3. V (1,0,1)=1
4. V (1,1,1)=0

La tercera entrada es fija. El vector de aprendizaje sería $W = (w_1, w_2, -c) = (w_1, w_2, 0)$. Así pues, las entradas y el vector de aprendizaje son elementos de $\mathbb{R}^3$. Además -c (que es cero) nos proporciona el punto en el que la recta de "clasificación" corta al eje vertical.

Podemos empezar a detallar el algoritmo de aprendizaje de un perceptrón. Supondremos que el conjunto de entradas $Input \subseteq \mathbb{R}^{n-1}$ es linealmente separable. Entonces digamos que nuestro conjunto de entradas es $I \subseteq \mathbb{R}^n$. De acuerdo a la definición de conjuntos linealmente separables, tenemos dos conjuntos de puntos P y N en R^n tales que:

1. $I = P \cup N$
2. $P \cap N = \emptyset$

3. Hay una recta que nos separa nuestras entradas que son de una región u otra, es decir $\exists W \epsilon \mathbb{R}^n((\forall X \epsilon\ P(X \cdot W > 0)) \wedge (\forall X \epsilon N(X \cdot W \leq 0)))$.

Dispondremos de una función F que nos modela el hecho de que **sí** conocemos, a priori, cómo clasificar una determinada entrada. El conjunto P debe contener a los "positivos" (los que van a quedar "arriba" de la recta de clasificación) y N de "negativos" (los que van a quedar "debajo" de la recta de clasificación).

$$F(X) : \mathbb{R}^n \to \mathbb{R}$$

$$F(X) = \begin{cases} 1 & si\ X \in P \\ 0 & si\ X \in N \end{cases} \tag{3.5}$$

Nuestra neurona tendrá asociada la siguiente función:

$$V(X) : \mathbb{R}^n \to \mathbb{R}$$

$$V(X) = \begin{cases} 1\ X \cdot W \geq 0 \\ 0\ X \cdot W < 0 \end{cases} \tag{3.6}$$

Con las definiciones antes mencionadas tenemos que nuestra respuesta deseada viene dada por la función F; la respuesta real de la neurona viene dada por la función V. Añadiremos un contador de las veces en que hemos modificado el vector W, el cual se incorpora como un subíndice t, por lo que la n-ésima ocasión en que hemos modificado el vector W se denota como W_n (W_0 puede ser generado aleatoriamente).

3.7. El algoritmo de aprendizaje para un perceptrón

Si nos hemos percatado que para cierta entrada X nuestra neurona ha clasificado mal entonces nos vemos en la necesidad de modificar el vector W. El algoritmo sugerido por Rosenblatt, en el cual se ajusta el vector W_t corrigiendo el error de la respuesta de la neurona al "clasificar" el vector de entradas X, tiene como regla:

$$W_{t+1} = W_1 + \eta * (F(X) - V(X)) * X \tag{3.7}$$

Donde η es una constante llamada *razón de aprendizaje*. Su labor consiste en agilizar el proceso de encontrar la recta que nos separe adecuadamente el conjunto

I. Es posible que el valor de η sea dado por una función que dependa del tiempo. Para simplificar el modelo, diremos que es constante con valor 1. En el caso de que la respuesta arrojada por la neurona sea la deseada, el vector W_{t+1} es exactamente el mismo que W_t pues $F(X) = V(X)$. Para no ser redundantes, **modificaremos el vector W con dicho algoritmo únicamente en el caso de ser necesario**.

3.7.1. Interpretación geométrica del algoritmo de aprendizaje

Dado que el valor de la expresión $F(X) - V(X)$ solo puede tomar los valores de 1 o -1, si $F(X) - V(X) \neq 0$ se puede decir que el vector W_{t+1} se va a "acercar" o "alejar" más del vector X con respecto del vector W_t para clasificarlo adecuadamente. Esta situación puede observarse en la imagen 3.5 gracias a la interpretación de la suma o resta de vectores con la ley del paralelogramo.

Supongamos que se ha hecho una mala clasificación y por ende se procede a ajustar el vector de pesos W_t. Si $X \epsilon P$ entonces $W_{t+1} = W_t + (1) * (1-0) * X = W_1 + X$ pues $F(X) - V(X) = 1 - 0$. En cambio, si $X \epsilon N$ entonces $W_{t+1} = W_t + (1) * (0-1) * X = W_1 - X$, pues $F(X) - V(X) = 0 - 1$.

Con la intención de fijar ideas, supongamos que tenemos un vector de entradas $X \in \mathbb{R}^2$ el cual además debería estar en la parte superior de nuestra recta de criterio (el vector fue mal clasificado). Dado que $X \in P$ y fue mal clasificado, tenemos que ajustar el vector W_1 de la siguiente manera:

$$W_2 = W_1 + X$$

Debemos recordar la interpretación geométrica de la suma de vectores (ley del paralelogramo) a la hora de sumar los vectores W_1 y X.

Suponiendo que $X \in P$, en el esquema mostrado en la imagen 3.5 el vector$X$ estaba mal clasificado pues se encontraba debajo de la recta que es tangente al vector W_1 . Una vez que hemos obtenido el vector W_2, nos fijamos que la recta perpendicular a W_2 (la recta de criterio en azul) ahora pasa por debajo del vector X, lo cual significa que ha quedado correctamente clasificado. Como se decía al principio de ésta sección, el vector W_2 se "acerca" más a X que W_1 (en lo que se refiere al ángulo).

Es posible que cada ajuste desacomode los vectores que ya habían sido correctamente clasificados e incluso que el ajuste no sea suficiente como para catalogar adecuadamente al vector que queríamos clasificar correctamente si únicamente realizamos el proceso una sola vez. Lo mejor es tomar un vector $X_1 \in I$, ajustar si es necesario, y después tomar otro vector $X_2 \in I$ distinto de $X_1 \in I$ aún cuando éste último no haya quedado correctamente clasificado.

Recapitulando, el algoritmo es:

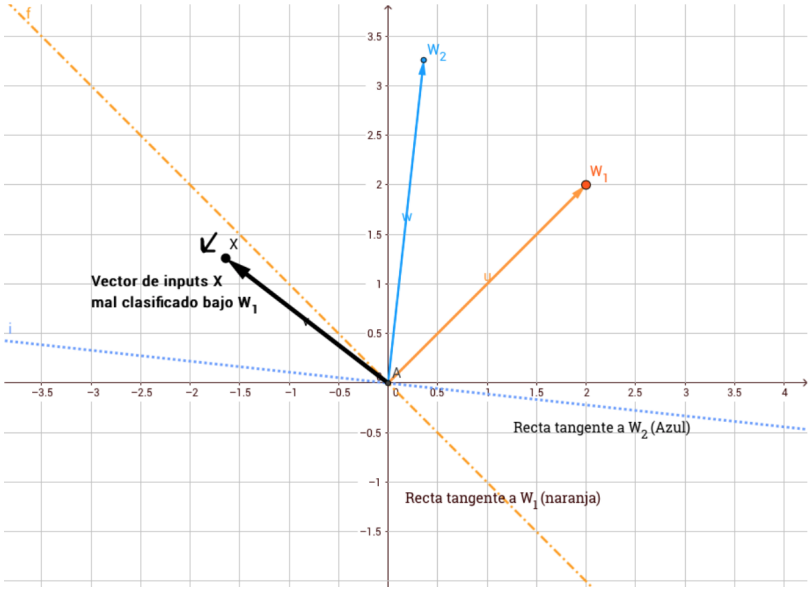


Figura 3.5: Interpretación geométrica de la regla de aprendizaje

1. *Comienzo*: Se genera el vector $W_0 = (w_1^0, ..., w_{n-1}^0, w_n^0)$ donde $w_n^0 = -c$.
2. *Test*: Se toma $Y \in I$ cualquiera al azar.
   - Si $Y \in P$ y $W_t \cdot Y > 0$ ir a test.
   - Si $Y \in P$ y $W_t \cdot Y \leq 0$ ir a sumar.
   - $Y \in N$ y $W_t \cdot Y \leq 0$ ir a test.
   - $Y \in N$ y $W_t \cdot Y > 0$ ir a restar.
3. *Sumar*: $W_{t+1} = W_t + X$ y $t = t + 1$, ir a test.
4. *Restar*: $W_{t+1} = W_t - X$ y $t = t + 1$, ir a test.

La notación w_i^t representa la coordenada i-ésima del vector W en la iteración t-ésima del algoritmo.

3.7.2. Prueba de la convergencia del algoritmo

Necesitamos asegurar que en algún momento vamos a terminar de ajustar los vectores W_t. Dado que a priori sabemos que el conjunto I es linealmente separable, afirmamos que existe al menos un $W^* \in \mathbb{R}^n$ que nos proporciona una recta que clasifica como queremos a los vectores del conjunto I.

Proposición: Si los conjuntos P y N son finitos y linealmente separables, el algoritmo de aprendizaje del perceptrón actualiza el vector W_t en un número finito de

pasos. Es decir, si los vectores en P y N son utilizados en el algoritmo, se encontrará un vector W_t después de un *número finito de pasos* que separa a los dos conjuntos P y N.

Prueba

Sea M el número de ajustes o errores que hemos realizado con el algoritmo de entrenamiento del Perceptrón. Además sea W^* el vector que nos permite separar el conjunto de puntos P y N.

PD. $M < \infty$

Para simplificar la prueba usaremos los siguientes supuestos y observaciones:

- Podemos formar un nuevo conjunto $P' = P \cup N^{-1}$ con $N^{-1} = \{-X : X \in N\}$. Este conjunto simplificará la prueba.
- Todos los ejemplos de entrenamiento X son tales que $X \neq \vec{0}$ porque su última entrada siempre es 1 (véase la adecuación al perceptrón que se muestra en la imagen 3.4).
- Podemos asumir que cualquier ejemplo de entrenamiento $X \in P'$ cumple que $\|X\| = 1$, es decir, todos los vectores de entrenamiento están normalizados. Nótese que al normalizar los vectores de entrenamiento no alteramos la labor de un vector de aprendizaje W . En efecto, si W es un vector de aprendizaje y $W \cdot X > 0$ entonces $W \cdot \frac{X}{\|X\|} > 0$ [2].
- Podemos asumir que el vector W^* que nos permite separar ambas regiones es tal que $\|W^*\| = 1$ (está normalizado). De no ser el caso, podemos normalizarlo sin alterar su capacidad de separar adecuadamente los ejemplos de entrenamiento. En efecto, si $W^* \cdot X > 0$ entonces $\frac{W^*}{\|W^*\|} \cdot X > 0$.
- Podemos suponer que todos los vectores $X \in P' = P \cup N^{-1}$ se encuentran clasificados como "positivos", es decir, se encuentran en la parte positiva del hiperplano con respecto de W^*. Lo anterior se debe a que, por la hipótesis de separabilidad lineal, si $X \in N$ entonces $X \cdot W^* < 0$. Multiplicando por -1 la desigualdad anterior se tiene que $-X \cdot W^* > 0$, es decir, $-X \in N^{-1} \subseteq P'$.
- Podemos asumir que el algoritmo comienza su labor con el vector de ceros en $\mathbb{R}^n$, es decir $W_0 = \hat{0}$.

Sea $\delta = min\{W^* \cdot P > 0 : \forall P \in P'\} > 0$. Podemos definir δ porque P' es finito y el vector W^* deja a todos los vectores en P' en la parte "superior"del hiperplano (véase las observaciones anteriores).

[2] Si $X \neq 0$, $\|X\| > 0$.

Observación 1: $W_{t+1} \cdot W^* \geq W_t \cdot W^* + \delta$.

1. $X \in P'$..... Hipótesis.
2. $X \cdot W_t \leq 0$..... Hipótesis, supondremos que X fue mal clasificado con W_t. Por lo que
3. $W_{t+1} = W_t + X$..... De 1 y 2, aplicando el algoritmo visto en 3.7.1. De tal manera que
4. $W_{t+1} \cdot W^* = (W_t + X) \cdot W^* = W_t \cdot W^* + X \cdot W^*$..... De 3, sustituyendo. Pero
5. $X \cdot W^* \geq \delta$..... Definición de δ. Finalmente
6. $W_{t+1} \cdot W^* \geq W_t \cdot W^* + \delta$ ■..... De 4 y 5.

Observación 2: $\|W_{t+1}\|^2 \leq \|W_t\|^2 + 1$.

Esto significa que cada error cometido por el algoritmo podría hacer crecer la norma del siguiente vector de pesos, pero no mucho más que que la cantidad indicada a la derecha de la desigualdad. En efecto:

1. $X \in P'$..... Hipótesis.
2. $W_t \cdot X \leq 0$..... Hipótesis, supondremos que X fue mal clasificado con el vector de pesos W_t. Por lo que:
3. $W_{t+1} = W_t + X$..... De 2, aplicando el algoritmo visto en 3.7.1. Recordando que
4. Si $Y \epsilon \mathbb{R}^n$ entonces $\|Y\|^2 = Y \cdot Y$..... Propiedades de la norma usual. Se tiene que
5. $\|W_{t+1}\|^2 = (W_t + X) \cdot (W_t + X) = \|W_t\|^2 + 2W_t \cdot X + \|X\|^2$..... Aplicando 4 a 3 y usando propiedades del producto punto y la definición de la norma usual. Luego
6. $\|W_{t+1}\|^2 \leq \|W_t\|^2 + \|X\|^2$.... De 2 y 5, se está restando una cantidad negativa a $\|W_{t+1}\|^2$. Recordando que
7. $\|X\| = 1$..... De 1 (véase las observaciones preliminares). Se tiene finalmente que
8. $\|W_{t+1}\|^2 \leq \|W_t\|^2 + 1$ ■..... De 6 y 7.

Con estas observaciones podemos concluir el resultado.

Supongamos que hemos actualizado $M \in \mathbb{N}$ veces el vector de pesos, con $M \geq 1$.

1. $W_{t+1} \cdot W^* \geq W_t \cdot W^* + \delta$..... Por observación 1.
2. $\|W_{t+1}\|^2 \leq \|W_t\|^2 + 1$..... Por observación 2.
   Después de M ajustes en nuestro algoritmo, tenemos que
3. $W_{M+1} \cdot W^* \geq M\delta$..... Aplicando inducción en 1 (Recordar que $W_0 = \hat{0}$).
4. $\|W_{M+1}\|^2 \leq M$..... Aplicando inducción en 2 (Recordar que $W_0 = \hat{0}$).
   Empero
5. Si $X, Y \epsilon \mathbb{R}^n$ entonces $|X \cdot Y| \leq \|X\| * \|Y\|$.....Desigualdad de Cauchy-Schwarz en $\mathbb{R}^n$.
   Luego
6. $|W_{M+1} \cdot W^*| \leq \|W_{M+1}\| * \|W^*\|$..... Aplicando 5 a W_M y W^*.
   Pero
7. $\|W^*\| = 1$..... (véase las observaciones liminares).
   De tal manera que
8. $|W_{M+1} \cdot W^*| \leq \|W_{M+1}\|$..... De 6 y 7.
   Es decir
9. $|W_{M+1} \cdot W^*|^2 \leq \|W_{M+1}\|^2$..... De 8, elevando al cuadrado ambos miembros.
   Además
10. $M^2 * \delta^2 \leq |W_{M+1} \cdot W^*|^2 = (W_{M+1} \cdot W^*)^2$..... De 3, elevando al cuadrado.
    Así
11. $M^2 * \delta^2 \leq \|W_{M+1}\|^2$..... De 9 y 10.
    Luego
12. $M^2 * \delta^2 \leq M$..... De 4 y 11.
    Finalmente
13. $M \leq \left(\frac{1}{\delta^2}\right)$ ■..... De 12, recordando que $M \geq 1$.

Lo anterior muestra que el número de ajustes no puede ser mayor que cierto valor fijo determinado por la norma del vector W^* y de la cantidad de entradas que hay en P' (el cual contiene todos los datos de entrada que se deben clasificar).

3.7.3. La función error de una neurona

Se ha mencionado que una neurona se puede interpretar como una función que aproxima a otra función, lo cual hace necesario definir una medida del error. Haremos un pequeño cambio en la notación. **Usaremos a letra t como la salida deseada de la red neuronal.** Dado un vector de entrada $X \epsilon \mathbb{R}^n$ de entrenamiento queremos que la neurona arroje una respuesta "muy parecida" a un vector predefinido $t \epsilon \mathbb{R}^m$ (del anglicismo *target*).

En notación, dada una función que representa el *output* o salida de una neurona: $V(X) : \mathbb{R}^2 \to \mathbb{R}$

Esta salida debe cumplir que:

$V(X) \approx t$ (donde $t \epsilon \mathbb{R}^m$)

Cambiaremos la notación $V(X)$ por la salida del vector $X \epsilon \mathbb{R}^n$, es decir $O(X)$ (de *output*). Cuando se entienda el contexto, podemos llamar simplemente la salida O de la neurona. La notación $V(X) \approx t$ la podemos interpretar como "el valor de salida está muy cerca del *target* u objetivo deseado". La noción de cercanía depende de la noción de una norma en un espacio vectorial. Para fines prácticos usaremos la norma usual en el espacio vectorial $\mathbb{R}^n$.

Finalmente, para n vectores de entrenamiento y misma cantidad de objetivos, definimos la función error de una neurona con la norma usual en $\mathbb{R}^m$ como:

$$E(W, X_i) = \frac{1}{2} * \sum_{i=1}^{n} \|O_i - t_i\|^2 \tag{3.8}$$

Donde cabe aclarar que $V(X_i) = O_i \in \mathbb{R}^m$ y $i \in \{1, 2, 3, ..., n\}$. Además de que la función E depende del vector de pesos y el ejemplo X_i.

3.7.4. Red neuronal artificial

Podemos construir un modelo en el que se involucren dos o más neuronas. Las entradas y salidas de estas neuronas van a interactuar de diversas maneras. La imagen mostrada en 3.6 simplifica la explicación de este modelo.

Cada nodo representa una neurona como las descritas anteriormente. Hay nodos que reciben las entradas a la red neuronal. Cada señal viaja desde los nodos denominados como capa de entrada hacia la capa uno a través de las aristas. Dentro de los nodos se ejecuta la función de activación, generando una nueva señal que se "propaga" a través de las aristas de salida. El "flujo de señales" va de izquierda a

derecha, lo cual se muestra en la imagen 3.7.

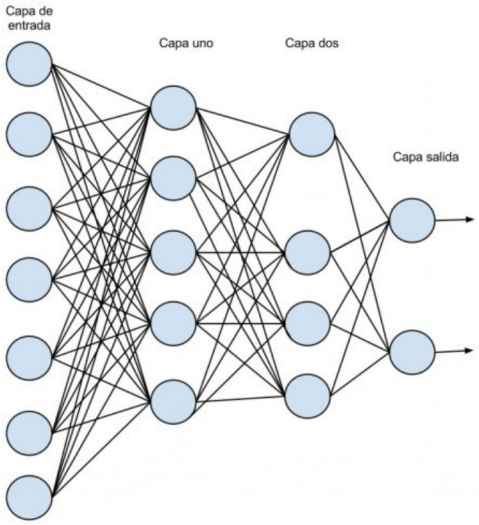


Figura 3.6: Grafo representando una red neuronal

Por simplicidad, consideraremos que cada señal que viaja a través de las aristas llega al mismo tiempo a cada nodo de la capa uno. De la misma manera cada señal que parte de los nodos de la capa uno a los nodos de la capa dos llega al mismo tiempo, siguiendo la trayectoria mostrada por las aristas del grafo. De los nodos de la capa de salida se emiten las salidas de la red neuronal.

En el ejemplo de red neuronal mostrado en la imagen 3.6, las capas uno y dos se conocen como capas ocultas. Un modelo de red neuronal puede incluir una o más capas ocultas.

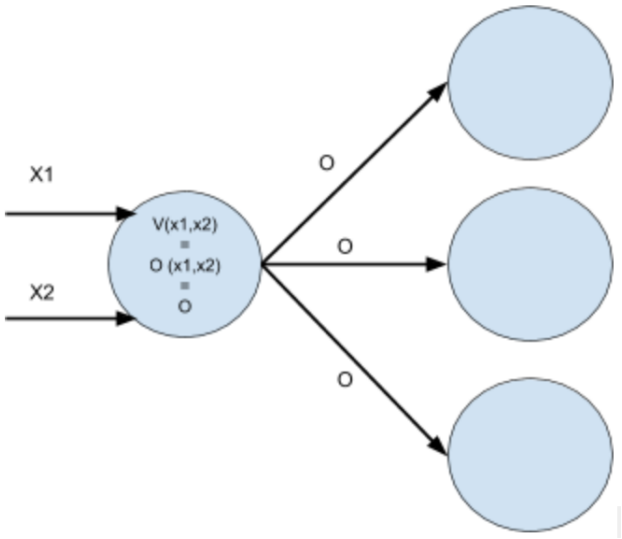


Figura 3.7: Grafo mostrando el flujo de datos en una neurona

3.7.5. Aprendizaje de una red neuronal y función error

De manera similar al procedimiento que usamos para el aprendizaje de una neurona, ajustaremos los pesos de las conexiones entre neuronas de la red neuronal para mejorar la aproximación a la respuesta deseada. Usaremos el enfoque utilizado en [10]. Pocos son los cambios realizados a ese enfoque y puede consultarse para profundizar en el método.

Consideremos un *conjunto de entrenamiento* $\{(X_1, t_1), ..., (X_p, t_p)\}$ en el que $X_i \in \mathbb{R}^n$ (entradas) y $t_i \in \mathbb{R}^m$ (del anglicismo *target*) con $i \in \{1, ..., p\}$. La red al procesar cada entrada arroja un *output* que denotaremos como O_i para $i \in \{1, ..., p\}$ (la salida también es un vector). Es decir, $O_i = V(X_i)$ donde la función V representa la función matemática que recibe como argumentos los vectores $X_i \in \mathbb{R}^n$ y devuelve valores $O_i = V(X_i) \in \mathbb{R}^m$. La función $V(X)$ es una composición de todas las funciones asociadas a las neuronas que forman la red neuronal y lo que buscamos es que $O_i = V(X_i) \approx t_i$; basta pedir entonces que para $\epsilon > 0$ suficientemente pequeño se cumpla que:

$$E = \frac{1}{2} * \sum_{i=1}^{n} \|O_i - t_i\|^2 \leq \epsilon \tag{3.9}$$

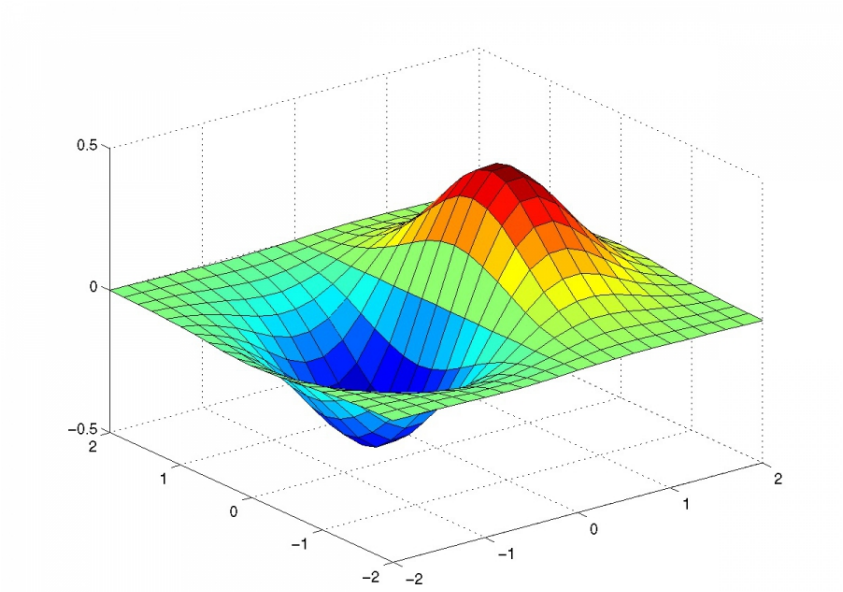


Figura 3.8: Ejemplo de una función de error en varias variables. El mínimo global está señalado en azul.

Como la red neuronal es una función de varias variables, el problema se limita a usar herramientas del cálculo multivariado para minimizar la función error

presentada. Esta función depende de los pesos de las conexiones entre las neuronas. En la imagen 3.8 se muestra un ejemplo de función multivariada.

Una vez que hemos minimizado la función error para el conjunto de entrenamiento podremos presentarle a la red entradas cualesquiera y esperar que responda correctamente o generalice adecuadamente.

3.7.6. Extensión de la red neuronal

Para calcular la función error extenderemos la red neuronal con algunos nodos extras.

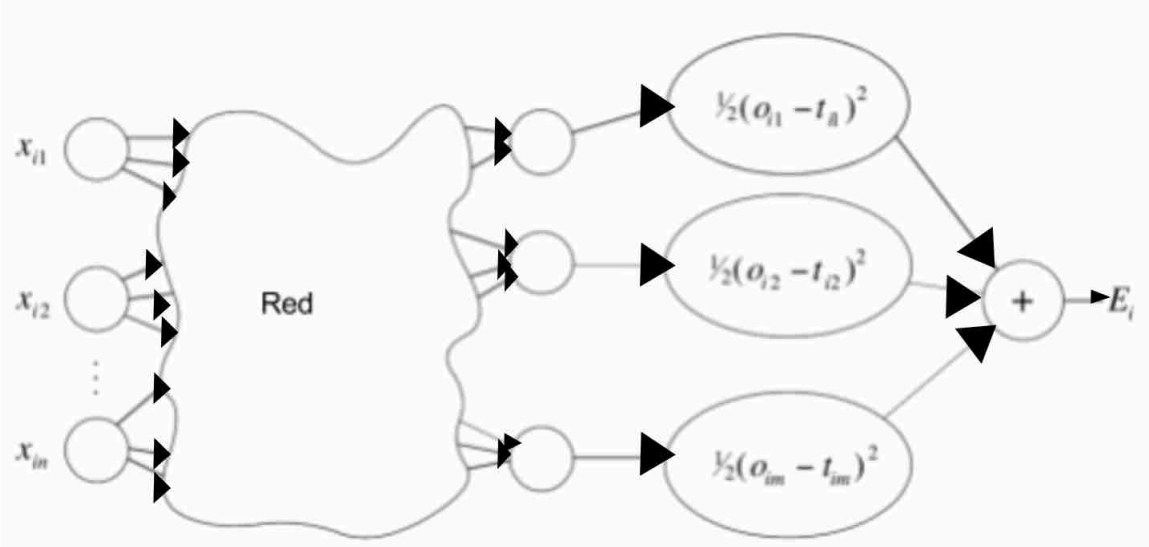


Figura 3.9: Red neuronal extendida. **El flujo de la información es de izquierda a derecha.**

La imagen 3.9 muestra un esquema del funcionamiento de la red neuronal. Supongamos que se introduce el vector de entrenamiento $X_i = (x_{i1}, ..., x_{in})$ cuya respuesta debería ser $t_i = (t_{i1}, ..., t_{im})$. Cada uno de las salidas de la red neuronal está conectada a un nodo que calcula la función $\frac{1}{2} * (o_{ij} - t_{ij})^2$ que es la diferencia entre la respuesta real y la respuesta deseada. Finalmente estos m nodos están conectados a un solo nodo que calcula la suma de ellos y arroja la salida que en la imagen es denotado como E_i.

Se realiza el mismo procedimiento para cada par de entrenamiento (X_i, t_i) con $i \in \{1, ..., k\}$ (es decir, hay k vectores de entrenamientos). Podemos unir la salida de cada red neuronal extendida en un solo nodo que calcula $E_1 + ... + E_k$ y arroja esta suma como salida. Dicha salida es $E = \frac{1}{n} * \sum_{i=1}^{n} \|O_i - t_i\|^2$.

3.7.7. El gradiente de la función error

El aprendizaje en la red neuronal consiste en corregir los pesos de manera iterativa para reducir el error. Para ello hacemos uso del gradiente[3]. Dado que el error E depende de la configuración de pesos $w_1, ..., wk$, el gradiente es una función que está dada como:

$$\nabla E(x_1, ..., x_n) = (\frac{\partial E}{\partial w_1}(x_1, ..., x_n), ..., \frac{\partial E}{\partial w_k}(x_1, ..., x_n)) \tag{3.10}$$

3.7.8. Actualización de los pesos de la red neuronal

Siendo k el número de aristas de la red neuronal, las cantidades $\frac{\partial E}{\partial w_i}$ nos proporcionan el incremento o decremento de los pesos denotado por:

$$\Delta w_i = -\gamma * \frac{\partial E}{\partial w_i} \quad para \quad i \in \{1, ..., k\} \tag{3.11}$$

Nótese que la manera en que actualizamos los pesos de la red neuronal es similar a la manera en que se actualizan los pesos en un perceptrón.

En la ecuación 3.11, γ representa la intensidad con la que deseamos que el ajuste de los pesos se lleve a cabo. Si el valor de γ es pequeño, los cambios podrían ser imperceptibles; si es grande, podría ser contraprudecente cuando el error cometido por la red neuronal es casi satisfactorio. En algunos casos es posible hacer de γ una variable que dependa del número de iteraciones o de cualquier otro parámetro. Por comodidad asumiremos que es constante.

La actualización de los pesos nos va a permitir encontrar la configuración que minimiza el error. Si bien el cálculo de las parciales nos indica la dirección en la que debemos desplazarnos para hallar el mínimo, no se trata del procedimiento de derivar e igualar a cero para hallar el mínimo de la función E. Resolver las ecuaciones en la que las parciales se anulan puede resultar una tarea compleja; peor aún, podríamos no disponer de una expresión analítica para la solución. En caso de que el sistema de ecuaciones sea factible de resolver podemos aplicar esa metodología sin problema.

[3]El gradiente de una función de varias variables es un vector en el que cada entrada es la derivada de la función con respecto de cada variable (derivada parcial).

3.7.9. El cálculo del gradiente de la función error

Buscamos calcular un gradiente que depende de los pesos de las conexiones entre los nodos. Para ello construiremos una red neuronal que nos proporcione dicho cálculo. Calcular las derivadas parciales de la función error con respecto de cada peso w_i sería muy ineficiente pues la red es una composición de funciones y la cantidad de pesos que hay en una red puede ser muy grande. En su lugar utilizaremos un método, que si bien implica cálculo de parciales, el costo computacional disminuye considerablemente.

Dado que la salida de la red neuronal extendida es resultado de una composición de funciones de los nodos, en nuestros cálculos jugará un papel importante la composición de funciones y la regla de la cadena.

Asumiremos que cada arista o conexión de una red neuronal puede transmitir información en ambos sentidos. Recorrer la red de izquierda a derecha nos proporcionará el error; de derecha a izquierda proporcionará el gradiente de la función error. Se asumirá que se recibe una entrada $x \in \mathbb{R}$. La derivada de una función de activación siempre se hará con respecto de la entrada x, por lo que f' denotará $\frac{\partial f}{\partial x}$.

3.7.9.1. Derivadas de la función de activación de un nodo

Con el siguiente procedimiento intentamos que cada nodo calcule exactamente una sola función y no dos como lo hacía el perceptrón. Recordemos que una sola neurona recibía las entradas, las sumaba (aplicaba la función suma) y posteriormente comparaba el valor obtenido con otro valor θ. Si la suma era mayor que θ, la neurona arrojaba un valor α (en el caso del perceptrón presentado, esa respuesta es 1); en caso contrario arrojaba un valor β (0 en el caso del perceptrón).

Cada nodo será dividido en una parte izquierda y en una parte derecha. El lado derecho del nodo evaluará la función de activación a partir de la entrada recibida como lo hace normalmente. Sin embargo, el lado izquierdo calculará la derivada de la función de activación con respecto de la variable de interés que queremos obtener, es decir, si denotamos la función de activación del nodo como f, podemos estar interesados en alguna derivada parcial $\frac{\partial f}{\partial w_i}$ con $i \in \{1, ..., k\}$. La imagen 3.10 proporciona una mejor explicación de lo que habremos de incorporar en cada nodo de la red.

Haremos además otra adecuación a la red. Separaremos de cada nodo la función que suma las entradas que recibe el nodo (*función de integración*). Las entradas serán recibidas por un nodo cuya función consiste en sumar las entradas. La salida de este nodo es la suma de las entradas recibidas y será transmitido al nodo que calcula la función de activación. La imagen 3.11 explica mejor el procedimiento.

Cuando realizamos esta separación, el nuevo nodo, cuya función es calcular

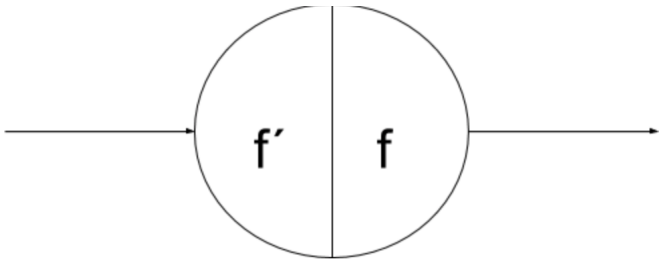


Figura 3.10: División del modelo de neurona. La parte derecha permanece sin cambios calculando la función de activación con respecto del *input*. La parte izquierda calcula la derivada de la función de activación con el *input* recibido.

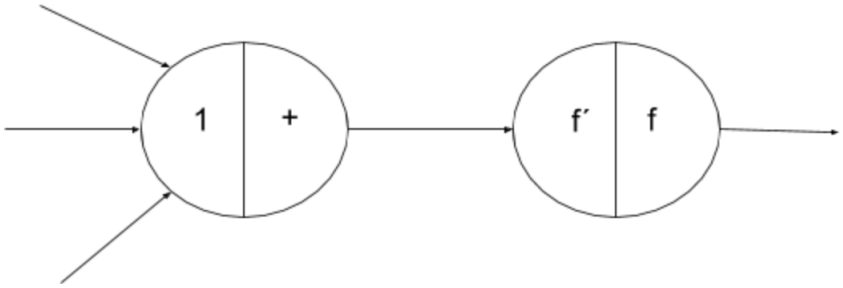


Figura 3.11: Separación de la función de activación y de la función de integración de un nodo.

la suma de las entradas, también estará dividido en su parte izquierda y su parte derecha. La parte derecha calcula la suma de las entradas recibidas del nodo original y la parte izquierda calculará la derivada parcial de la función suma. La derivada parcial de la función suma con respecto a una variable, digamos w_i, es 1, por lo que la salida del lado izquierdo es la constante 1.

Así pues cada nodo de la red neuronal original tendrá el funcionamiento que se explica a continuación.

El primer paso considera el flujo de la información de izquierda a derecha *feed-forward step*.

Cuando se reciben las entradas de izquierda a derecha (*feed-forward step*), las entradas recibidas son sumadas por el nodo que tiene como función la función suma (en la figura 3.11). En dicho nodo, el lado izquierdo guarda el valor uno y el lado derecho guarda la suma de las entradas. Sólo el valor del lado derecho es transmitido hacía la derecha siguiendo la arista. Ese valor (la suma de las entradas) es utilizado como argumento por el nodo que calcula la función de activación f y su derivada $\frac{\partial f}{\partial w_i}$ (en la imagen 3.10 se usa f prima). Los valores numéricos de $\frac{\partial f}{\partial w_i}$ y f son guardados por el nodo pero sólo el derecho es transmitido hacia la derecha siguiendo las aristas.

El segundo paso considera el flujo de los errores cometidos por la red neuronal

de derecha a izquierda (*backpropagation step*), el cual consiste en recorrer la red hacia atrás utilizando los valores almacenados a la izquierda de cada nodo durante el paso *feed-forward step*.

Se analizan ambos pasos para ciertos subgrafos elementales a partir de los cuales se puede describir cualquier red neuronal. En cada uno de estos análisis es preferible que se acuda a las imágenes mostradas para un mayor entendimiento del procedimiento descrito.

3.7.9.2. Composición de funciones

Supongamos que tenemos el fragmento de red neuronal descrito por la imagen 3.12.

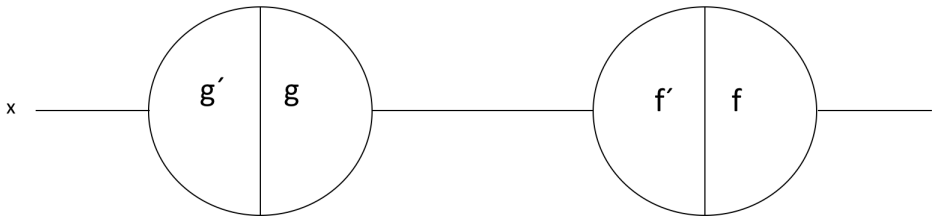


Figura 3.12: Grafo o red representando una composición de funciones

Feed-forward step: Se recibe una entrada $x \in \mathbb{R}$ en el primer nodo (de izquierda a derecha) el cual evalúa la función g' y g en x. Se guardan los valores $g'(x)$ y $g(x)$ en la parte izquierda y derecha del nodo respectivamente. Únicamente el valor numérico $g(x)$ es transmitido hacia la derecha siguiendo la arista (o las aristas) .

Del mismo modo, el valor $g(x)$ es recibido como entrada en el segundo nodo, el cual evalúa f' y f en $g(x)$. Se guardan los valores $f(g'(x))$ y $f(g(x))$ en la parte izquierda y derecha del segundo nodo respectivamente. Únicamente el valor numérico $f(g(x))$ es transmitido hacia la derecha siguiendo la arista (o las aristas) . En la imagen 3.13 se explica mejor este procedimiento.

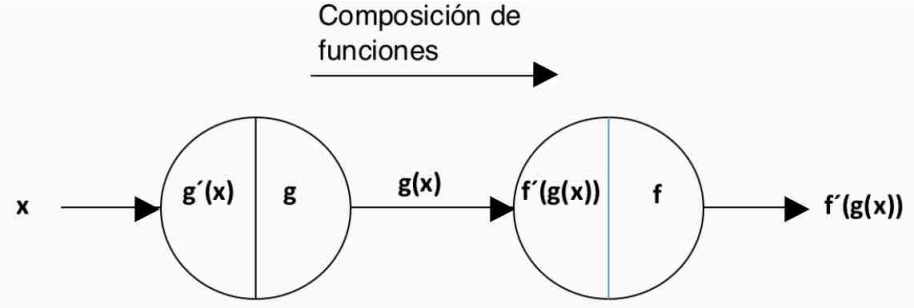


Figura 3.13: Resultado de recorrer la red hacia adelante o *feed-forward step*.

Backpropagation step: En esta etapa la red recibe una entrada que por simplicidad supondremos de 1 (podría ser un valor cualquiera $x \in \mathbb{R}$).

El valor 1 se propaga hacia la izquierda multiplicándolo por los valores que hay en la parte izquierda de los nodos. Al final, nos resulta la regla de la cadena: $f'(g(x)) * 1 = f'(g(x))$.

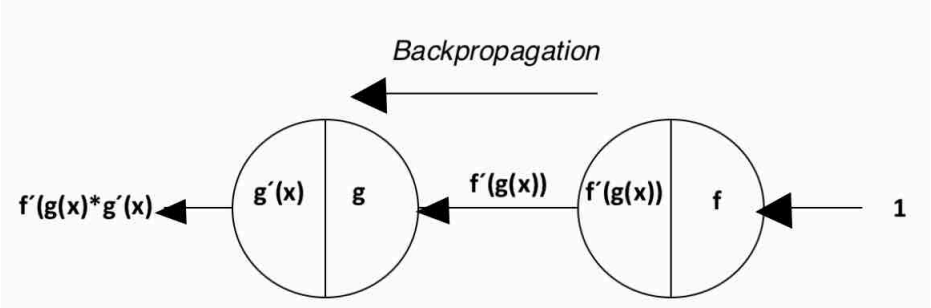


Figura 3.14: Resultado de recorrer la red hacia atras o *backpropagation step*.

3.7.9.3. Suma de funciones

Para encontrar la derivada de la suma de dos funciones f_1 y f_2 construimos una red como la que se muestra en la imagen 3.15.

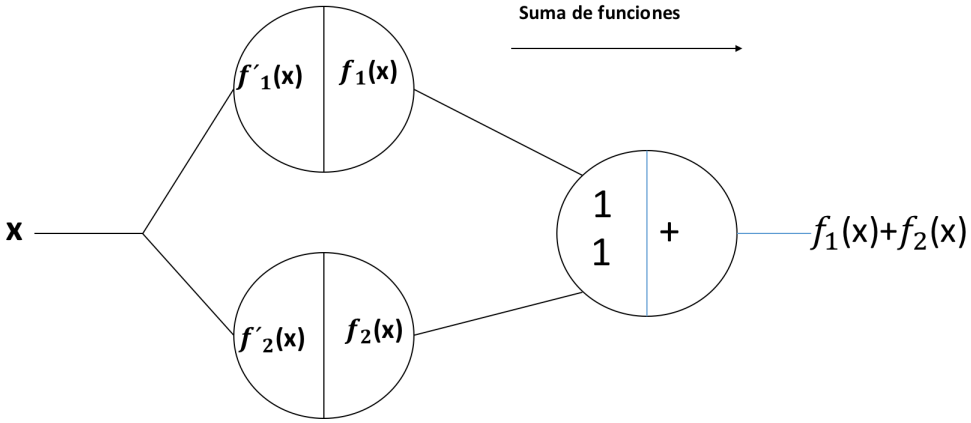


Figura 3.15: Resultado de recorrer la red hacia adelante o *feed-forward step* para la suma de funciones.

El nodo adicional en el esquema cumple la labor de sumar las dos funciones. Como ya hemos visto antes, la derivada de la función suma con respecto de cualquiera de las dos entradas que recibe es 1 (es decir, con respecto de las dos aristas que inciden en el nodo que se esquematiza en la figura 3.15), por lo que la parte izquierda de dicho nodo adicional es 1.

Feed-forward step: Leyendo la gráfica de izquierda a derecha (el sentido en el que se propagan los datos durante este paso), tenemos una entrada $x \in \mathbb{R}$ que viaja hacia la derecha y llega a los dos primeros nodos. Estos nodos calculan la función que tienen asignada (f_1 y f_2 respectivamente). La respuesta de cada nodo sigue su camino hasta que ambas inciden en el nodo encargado de sumar estas respuestas. Al final de este paso, obtenemos la suma de funciones deseada $f_1 + f_2$. Se muestra este procedimiento en la figura 3.15.

Backpropagation step: La red es alimentada con una entrada de 1 que viaja hacia la izquierda. Al llegar al primer nodo la entrada se multiplica por lo que se encuentra en la parte izquierda del nodo. Aquí notamos que hay dos valores guardados (ambos valen 1) los cuales serán multiplicados por 1. Se producirá la respuesta y viajará en cada arista hacia aquellos nodos cuyo valor almacenado en su lado izquierdo es f_1' y f_2'. Siguiendo las respectivas aristas, se llega a la bifurcación, la cual hace que las respuestas de los nodos se sume. Al final obtenemos la derivada de la suma de dos funciones, es decir $f_1' + f_2'$. Se muestra en la imagen 3.16 el procedimiento.

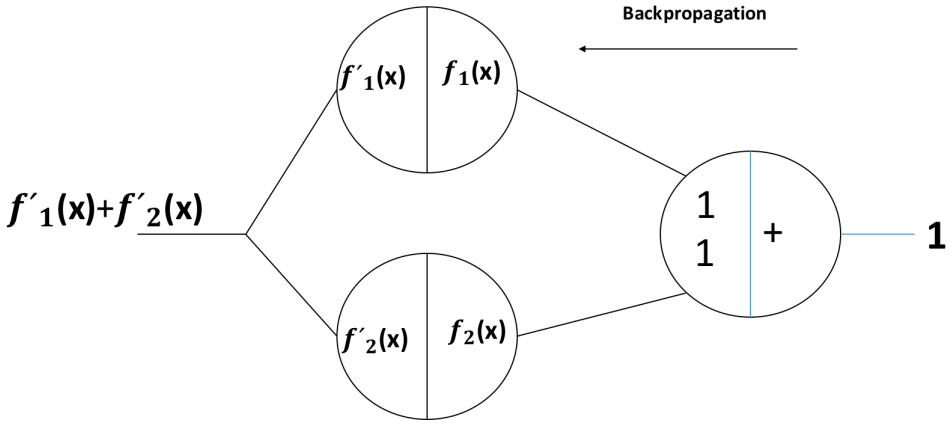


Figura 3.16: Resultado de recorrer la red hacia atras o *backpropagation step* para la suma de funciones.

Cabe mencionar que es posible sumar más de dos funciones. El procedimiento es completamente análogo.

3.7.9.4. Aristas

Feed-forward step: La información $x \in \mathbb{R}$ que viaja a través de la arista es multiplicada por el peso de la arista w. El resultado es $w * x$ (imagen 3.17).

Backpropagation step: Se introduce un valor 1 a la arista que se propagará de derecha a izquierda. Este valor es multiplicado por el peso de la aristas $w \in \mathbb{R}$. Al final, obtenemos el valor w el cual coincide con ser la derivada de la función $f(x) = w * x$ con respecto de x (imagen 3.18.

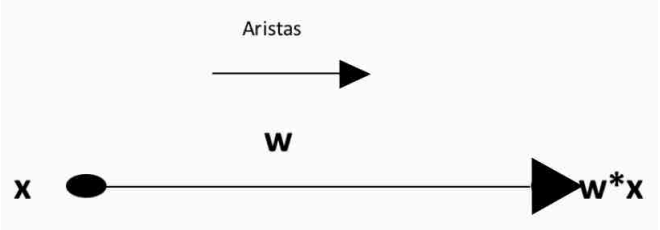


Figura 3.17: Resultado de recorrer una arista hacia adelante o *feed-forward step*.

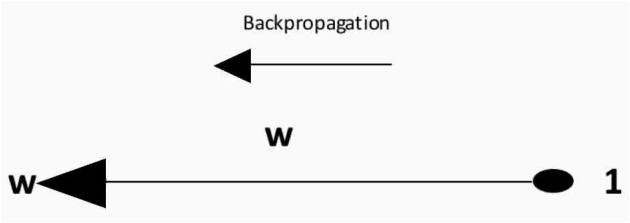


Figura 3.18: Resultado de recorrer una arista hacia atrás o *backpropagation step*.

3.7.9.5. Pasos del algoritmo de retropropagación (backpropagation algorithm)

Una vez que tenemos los elementos básicos que nos permiten construir una red neuronal que calcula el gradiente de la función error podemos mencionar los pasos del algoritmo de retropropagación o *backpropagation algorithm*. Se asumirá que las funciones de activación de cada nodo de la red neuronal original son funciones derivables. Además, se tiene un nodo inicial y un nodo final. El nodo final arroja como respuesta la aproximación a la función deseada. Denotemos a esta aproximación como $F(x)$. Esta función se conoce como la *función de la red*.

Algoritmo de retropropagación: Tomamos una red neuronal cuya función de red es F. Se aplican los procedimientos de la sección 3.7.9. Esta será la red sobre la cual implementaremos el algoritmo de retropropagación.

- Recorrido hacia adelante de la red o *feed-forward step*: La red es alimentada con la entrada $x \in \mathbb{R}$. Dicha entrada es propagada hacia adelante (derecha) y pasa por los nodos de la red. En cada nodo se calcula el lado izquierdo y el lado derecho (la derivada de la función de activación y la propia función de activación respectivamente). Hay que recordar que el valor de las derivadas son almacenados en los nodos pero no son propagadas hacia adelante durante este paso.
- Recorrido hacia atrás o *backpropagation step*: Se alimenta a la red con valor de 1 y se recorre la red de derecha a izquierda. Se siguen los procedimientos especificados en la sección 3.7.9. En esa sección se utilizó un 1 como entrada durante este paso, pero ahora la entrada puede ser una constante cualquiera

(resultado de cálculos anteriores posiblemente). No hay un cambio significativo en utilizar una constante cualquiera en vez de 1 y los procedimientos descritos en ese apartado son totalmente análogos. Al final obtenemos la derivada de la función de la red F con respecto de x.

A continuación se prueba que, en efecto, el algoritmo presentado calcula la derivada de la función de red F con respecto de x.

Proposición: El algoritmo de retropropagación calcula la derivada de F con respecto del input x.

Demostración (por inducción sobre el número de nodos de la red).

Como casos bases se toman todas los casos abordados en la sección 3.7.9.

Hipótesis inductiva: Asumimos que el algoritmo de retropropagación funciona para cualquier red con un número de nodos menor o igual a n, es decir, dado una red que posee n o menos nodos, es posible calcular la derivada de una función de red G con respecto de la variable x.

Consideremos la red neuronal con $n+1$ nodos que se muestra en la figura 3.19:

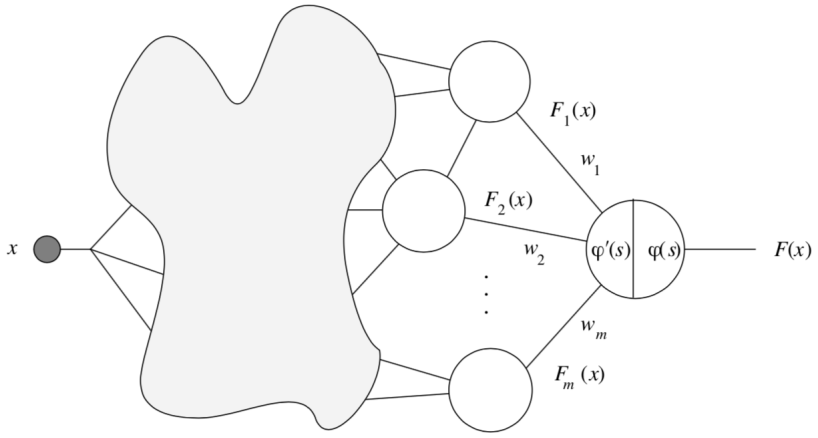


Figura 3.19: Red neuronal con $n+1$ nodos. La zona sombreada denota la presencia de más nodos.

Se alimenta a la red con la entrada x, al correr la red hacia adelante (de izquierda a derecha) o hacer el *feed-forward step* y se realizan los procedimientos señalados en la sección 3.7.9. El nodo final encargado de la salida total de la red arroja el valor F, es decir, arroja el valor de la función de red. Asumamos que hay m nodos conectados al último nodo cuyas funciones de activación son $F_1, ..., F_m$. En la imagen 3.19 llamamos a la función de activación del último nodo ϕ.

En la imagen 3.19 podemos darnos cuenta que $F(x) = \phi(w_1 * F_1 + ... + w_m *$

F_m). Derivando la función F con respecto de x obtenemos que $F\prime(x) = \phi\prime(s) * (w_1 * F\prime_1(x) + ... + w_m * F\prime_m(x))$ donde $s = w_1 * F_1 + ... + w_m * F_m$.

Ahora notamos que la subgráfica que incluye todos los posibles caminos entre el primer nodo de la red al nodo cuya salida es la función $F_1(x)$ define una subred cuya *función de red* es F_1 y la cual contiene n o menos nodos. Por la hipótesis de inducción que presentamos, es posible calcular $F\prime_1$ introduciendo un 1 en esa sub red y recorriendo hacia atrás la red (con los procedimientos señalados en 3.7.9). Lo mismo puede hacerse para los nodos cuyas salidas son $F_2(x), .., F_m(x)$.

De esta manera, la red nos proporciona el valor de $F\prime_1(x), .., F\prime_m(x)$. Si en vez de 1 hacemos recorrer hacia atrás el valor constante $\phi\prime(s) * w_1$ como entrada durante el paso en que recorremos la red hacia atrás o *backpropagation step*, obtendremos $w_1 * F\prime_1 * \phi\prime(s)$. Análogamente podemos obtener $w_1 * F\prime_1(x) * \phi\prime(s), ..., w_m * F\prime_m(x) * \phi\prime(s)$ para el resto de los nodos.

Al recorrer la red neuronal hacia atrás con la red completa (esto es, con el nodo cuya salida es $F(x)$), sumamos estos m términos y obtenemos $\phi'(s) * (F_1'(x) + F_2'(x))$, lo cual es la derivada de la función F evaluada en x como se requería demostrar cuando hay n+1 nodos ■.

En la anterior demostración hay que recordar que todos las entradas de un nodo son sumados para que después dicha suma pueda ser evaluada en la función de activación.

3.7.10. Generalizaciones del algoritmo de retropropagación

En el algoritmo anterior también es posible considerar funciones de activación de los nodos con más de un argumento. En este caso la parte izquierda de cada nodo de la red guarda todas las derivadas parciales de la función f con respecto de cada variable. La imagen 3.20 muestra un ejemplo de un nodo que recibe dos entradas para ser procesadas por su función de activación, la cual recibe dos argumentos x_1 y x_2. Estas entradas son entregadas a través de dos diferentes aristas o conexiones.

Al especificar que queremos alguna derivada parcial con respecto de una variable y recorrer la red hacia atrás, cada derivada parcial almacenada es multiplicada por la entrada recibida. Ese resultado es transmitido hacia la izquierda a través de la arista que le corresponde.

El algoritmo de retropropagación presentado también funciona cuando se añaden más *unidades de entrada*. Por ejemplo, en una red con dos unidades de entrada x_1, x_2 la función de red puede ser llamada $F(x_1, x_2)$. Ahora podemos calcular la derivada parcial de F con respecto de x_1 o x_2. El recorrido hacia adelante no presenta cambios, considerando que cada uno de los nodos debe ser una función de dos variables

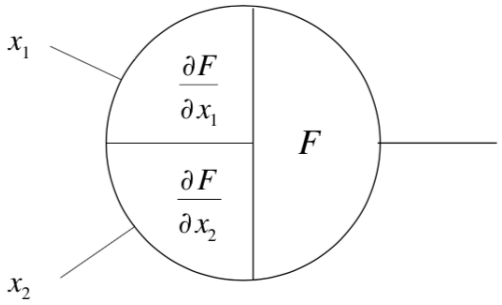


Figura 3.20: Derivadas parciales guardadas en un nodo.

en nuestro caso (aunque no sean usadas todas las variables en dichas funciones [4]). El llenado de los lados izquierdos de los nodos permanece sin cambios (las parciales con respecto de una variable pueden ser cero).

Al recorrer la red hacia atrás podemos identificar dos subredes: una consiste en todos los caminos que conectan un nodo de entrada (el recibe la entrada que denotamos como x_1) y el nodo de salida, y la red que consiste en todos los caminos que conectan el nodo de entrada que recibe la señal que denotamos como x_2 con el nodo de salida. Aplicando el algoritmo de retropropagación a cada una de las subredes obtenemos la derivada parcial de F con respecto de x_1 y x_2 respectivamente. Incluso podríamos calcular simultáneamente ambas parciales en la misma red gracias a las generalizaciones presentadas.

3.7.10.1. Aprendizaje a través del algoritmo de retropropagación

La sección anterior muestra una manera de calcular el gradiente de una función F cualquiera usando una red neuronal. Con esta herramienta podemos abordar el aprendizaje en redes neuronales de manera general. Estamos interesados en la función error definida en la sección 3.7.9, la cual depende del peso de las conexiones entre las neuronas o aristas de la red.

Aplicaremos el algoritmo de retropropagación descrito en la parte 3.7.9.5 a la red neuronal extendida (descrita en 3.7.6). Suponiendo que se han numerado los nodos de la red neuronal, nos fijaremos en alguna arista de la red, digamos w_{ij} que conecta al i-ésimo nodo con el j-ésimo nodo de la red. Esta arista puede ser vista como un canal de entrada de la subred formada por todos los caminos que empiezan con dicha arista y terminan en el último nodo de toda la red (vista de izquierda a derecha). Pongamos nuestra atención en esta subred.

Siendo o_i el valor almacenado en la parte derecha del nodo i-ésimo (y el cual es propagado hacía la derecha), la información introducida en esta subred definida en

[4] Considérese la función $F(x, y) = x$ por ejemplo

el párrafo anterior sería $o_i * w_{ij}$. Al echar a andar el algoritmo de retropropagación y recorrer la red hacia atrás se calcula el gradiente de la función E con respecto del *input* mencionado, es decir $\frac{\partial E}{\partial o_i w_{ij}}$. Al recorrer la red hacia atrás , realizando los pasos mencionados en la parte 3.7.9, o_i es tratado como una constante, por lo que:

$$\frac{\partial E}{\partial w_{ij}} = o_i * \frac{\partial E}{\partial o_i w_{ij}} \tag{3.12}$$

En conclusión los pasos a realizar durante el recorrido hacia atrás de la red son los mismos. Todas las subredes definidas por cada arista (o conexión entre neuronas) pueden ser consideradas simultáneamente pero se tiene que guardar en cada nodo i:

- La salida o_i del nodo (ya se había especificado esto en 3.7.9).
- El resultado acumulado hasta ese nodo i durante el proceso de recorrer hacia atrás la red. A dicho resultado lo llamamos *error de retropropagación*. Este resultado puede interpretarse como la parte del error en el que ha cooperado la arista o conexión.

Si denotamos el error de retropropagación en el j-ésimo nodo por δ_j, podemos expresar la derivada parcial de E con respecto de la arista w_{ij} como: $\frac{\partial E}{\partial w_{ij}} = o_i * \delta j$.

Una vez que todas las derivadas parciales de la función E han sido calculadas (con respecto de cada arista de la red extendida) podemos llevar a cabo el algoritmo del descenso por el gradiente añadiendo a cada peso w_{ij} el incremento :

$$\Delta w_{ij} = -\gamma * o_i * \delta j \tag{3.13}$$

Este paso en el que corregimos los pesos de la red es necesario para transformar el algoritmo de retropropagación en un algoritmo de aprendizaje para las redes neuronales.

3.7.11. El perceptrón multicapa

Dentro de las redes neuronales hay una arquitectura conocida como *perceptrón multicapa*. La organización de los nodos en capas constituye un perceptrón multicapa o una red neuronal *fully layer conected*. En la imagen 3.6 se muestra un perceptrón multicapa con dos capas ocultas que son llamadas capa uno y capa dos. Estas neuronas ocultas permiten que la red aprenda tareas complejas como clasificar

puntos de manera más precisa que con un hiperplano (o una recta) como se muestra en la figura 3.21.

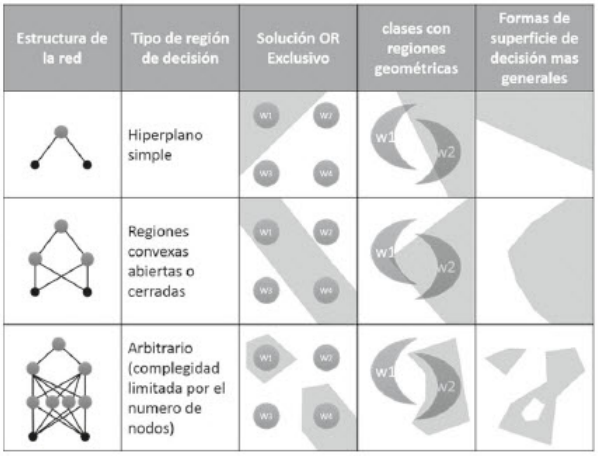


Figura 3.21: Varias neuronas organizadas en capas permiten generar regiones de decisión más complejas que dos semiplanos. [7]

En un perceptrón multicapa, cada neurona incluye una función de activación no lineal suave. No se permiten aristas entre los nodos de una misma capa pero en modelos más generales, por ejemplo las redes recurrentes, se pueden establecer conexiones incluso con nodos de capas anteriores. [5]

3.7.11.1. Aprendizaje en un perceptrón multicapa

El algoritmo de retropropagación presentado anteriormente es posible aplicarlo en cualquier red cuyos datos se propaguen hacía adelante (no necesariamente una estructura de perceptrón multicapa o red totalmente conectada). Sin embargo, es posible dar una expresión matricial para el caso de una red neuronal organizada por capas. Mostraremos una fórmula explícita para la actualización de pesos y mostraremos cómo esta actualización es llevada a cabo con procedimientos de álgebra lineal.

3.7.11.1.1. Red extendida del perceptrón multicapa Para simplificar la exposición usaremos un perceptrón multicapa con una sola capa oculta que consta de k nodos. El esquema que utilizaremos se muestra en la imagen 3.22:

Consideraremos una red con n nodos de entrada y m nodos de salida. Al peso de la arista entre la unidad i de la capa de entrada y la unidad j de la capa oculta será llamada $w_{ij}^{(1)}$. De la misma manera, al peso de la arista entre la unidad

[5] Puede consultarse más sobre estos modelos en el capítulo 10 de [13].

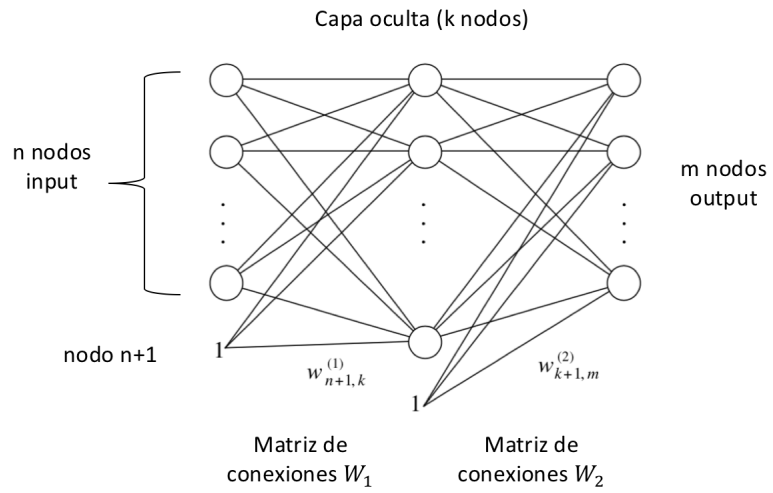


Figura 3.22: Esquema de un perceptrón multicapa con tres capas. Se presenta una capa oculta con k nodos.

i de la capa de oculta y la unidad j de la capa de salida será llamada $w_{ij}^{(2)}$. El valor umbral $-\theta$ de cada nodo es implementado como el peso de una arista adicional.

Los vectores de entrada son extendidos con una entrada extra cuyo valor será 1, y lo mismo se hace con los vectores de salida de la capa oculta.

El peso entre la constante 1 y la unidad j de la capa oculta es llamada $w_{n+1,j}^{(1)}$ y el peso entre la constante 1 de la capa oculta y el nodo de salida j es denotado como $w_{k+1,j}^{(2)}$.

Hay $(n+1)*k$ pesos entre los nodos de la capa de entrada y los nodos de la capa oculta. De igual manera, hay $(k+1)*m$ pesos entre los nodos de la capa oculta y los nodos de la capa de salida. Llamemos $\overline{W_1}$ a la (n+1) x k matriz cuyo componente $w_{ij}^{(1)}$ es i-ésimo renglón y la j-ésima columna.

De igual manera $\overline{W_2}$ denota la matriz de dimensión (k+1) x m con componentes $w_{ij}^{(2)}$. La barra en la notación para ambas matrices sirve para enfatizar que el ultimo renglón corresponde a los valores umbrales de los nodos de la capa oculta y la capa de salida. La matriz de pesos sin ese último renglón será necesaria en el paso cuando recorremos la red neuronal hacia atrás o *backpropagation step*. El vector de entrada de dimensión n que denotaremos como $o = (o_1, ..., o_n)$ es extendido a $\widehat{o} = (o_1, ..., o_n, 1)$.

Notamos que el nodo j-ésimo de la capa oculta recibe como entrada el valor que denotaremos como net_j y se expresa como:

$$net_j = \sum_{i=1}^{n+1} w_{ij}^{(1)} * \widehat{o_i} \tag{3.14}$$

La función de activación es una función matemática conocido como sigmoide [6] y la salida $o_j^{(1)}$ de esta unidad es:

$$o_j^{(1)} = s(\sum_{i=1}^{n+1} w_{ij}^{(1)} * \widehat{o_i}) \tag{3.15}$$

La excitación del nodo i-ésimo en la capa oculta puede ser calculada rápidamente con la operación matricial $\widehat{o}\overline{W_1}$ la cual nos arroja una matriz de dimensión 1xk, es decir un vector de k entradas. Simplemente nos fijamos en la entrada i-ésima de dicho vector.

Además, el vector $o^{(1)} = (o_1^{(1)}, ..., o_k^{(1)})$ cuyos componentes son las salidas de los nodos de la capa oculta están dados por:

$$o^{(1)} = s(\widehat{o}\overline{W_1}) \tag{3.16}$$

La función sigmoide s se aplica a cada entrada del vector. Haremos la convención de que la función es la misma aunque es posible considerar diferentes funciones de activación dependiendo de la entrada del vector.

La excitación de los nodos en la capa de salida es calculada usando el vector extendido $\widehat{o}^{(1)} = (o^{(1)}, ..., o^{(k)}, 1)$. La salida de la red es el vector de m entradas $o^{(2)} = (o_1^{(2)}, ..., o_m^{(2)})$ donde:

$$o^{(2)} = s(\widehat{o}^{(1)}\overline{W_2}) \tag{3.17}$$

Estas fórmulas pueden ser generalizadas para cualquier número de capas y permitir el cálculo del flujo de información en la red con operaciones matriciales.

3.7.11.1.2. Pasos del algoritmo La figura 3.23 muestra la red extendida que nos ayudará a calcular la función error.

Para simplificar la exposición del algoritmo alimentaremos a esta red con un vector de entrada $\widehat{o} = (o_1, ..., o_n, 1)$ el cual nos arroja una salida $t = (t_1, ..., t_m)$; es decir usaremos el par $(\widehat{o}, t)$ y podremos generalizar a p ejemplos de entrenamiento

[6] véase más sobre las funciones sigmoides en [6]

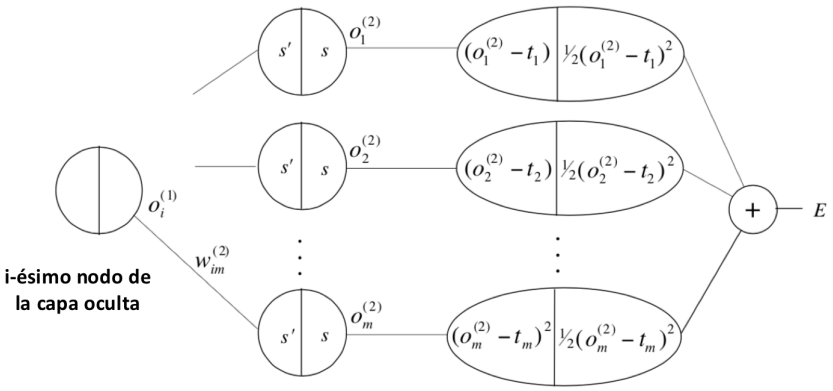


Figura 3.23: Red multicapa extendida usada para el cálculo de la función error.

después.

La red mostrada en 3.23 fue extendida con una capa adicional de nodos o neuronas. El lado derecho calcula la desviación cuadrática $\frac{1}{2} * (o_i^{(2)} - t_i)$ para el i-ésimo componente del vector salida de la red; el lado izquierdo guarda el valor $(o_i^{(2)} - t_i)$ que coincide con ser la derivada de la desviación cuadrática como ya se había especificado en secciones anteriores.

Cada nodo de salida de la red original calcula la función sigmoide s usando como argumento sus respectivas entradas y produce como salida el valor que denotamos en la imagen 3.23 como $o_i^{(2)}$. La suma de las desviaciones cuadráticas nos proporciona el error que denotamos en la imagen como E.

La función error para p pares de entrenamiento puede ser calculado creando p redes como la que se muestra en la imagen 3.23, una por cada par de entrenamiento, y sumando los salidas de todas estas redes.

Comenzamos seleccionando los pesos de aristas aleatoriamente. Después utilizamos el algoritmo de retropropagación para hacer las correcciones necesarias. A continuación se describe los pasos del algoritmo de manera general:

1. Recorrido hacia adelante de la red o *feed-forward step*.
2. Realizar el algoritmo de retropropagación hacia la capa de salida de la red.
3. Realizar el algoritmo de retropropagación en la capa oculta.
4. Actualización de los pesos de la red.

Se realizan todos los pasos descritos hasta que el valor de la función error es suficientemente pequeño.

3.7.11.1.3. Primer paso: recorrido hacia adelante El vector $\widehat{o} = (o_1, ..., o_n, 1)$ es presentado a la red. Los vectores $o^{(1)}$ y $o^{(2)}$ son calculados y almacenados. Las derivadas de las funciones de activación evaluadas son guardadas en cada unidad.

3.7.11.1.4. Segundo paso: retropropagación en la capa oculta Buscamos el primer conjunto de derivadas parciales $\frac{\partial E}{\partial w_{ij}^{(2)}}$. El camino que sigue el algoritmo de retropropagación desde el la salida de la red extendida (la que calcula el error) hasta los nodos de salida de la red original (es decir, la que no calcula la función error) es mostrado en la imagen 3.24.

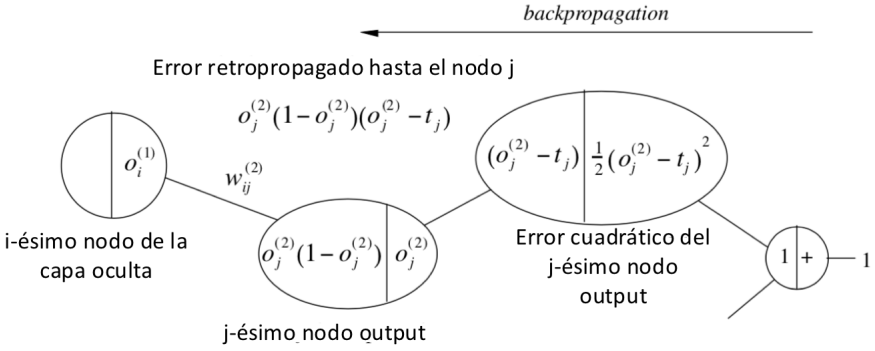


Figura 3.24: Camino del algoritmo de retropropagación hasta el nodo de salida j.

Recorriendo la red hacia atrás llegamos al error de retropropagación $\delta_j^{(2)}$ que es:

$$\delta_j^{(2)} = o_j^{(2)} * (1 - o_j^{(2)}) * (o_j^{(2)} - t_j) \tag{3.18}$$

Siendo la función de activación generalmente $s(x) = \frac{1}{1+e^x}$, la derivada de la función de activación es $s(x) * (1 - s(x))$. Si $s(x) = o_j^{(2)}$, obtenemos el resultado indicado.

Y la derivada parcial que buscamos es:

$$\frac{\partial E}{\partial w_{ij}^{(2)}} = [o_j^{(2)} * (1 - o_j^{(2)}) * (o_j^{(2)} - t_j)]o_i^{(1)} = \delta_j^{(2)} o_i^{(1)} \tag{3.19}$$

Recordemos que en este último paso consideramos el peso $w_{ij}^{(2)}$ como una variable y la entrada $o_i^{(1)}$ como constante.

3.7.11.1.5. Tercer paso: retropropagación a la capa oculta Ahora buscamos calcular las derivadas parciales $\frac{\partial E}{\partial w_{ij}^{(1)}}$. Cada nodo j en la capa oculta esta conectada a cada unidad q en la capa de salida con una arista cuyo peso es $w_{jq}^{(2)}$, con $q \in \{1, ..., m\}$ (es decir, el número de nodos en la capa *output*). El error retropropagado hasta el nodo j de la capa oculta debe ser calculado tomando en cuenta todos los posibles caminos hacia atrás como es mostrado en la imagen 3.25. Además, recordemos que ya hemos calculado el valor $\delta_j^{(2)}$ según lo expresamos en la ecuación 3.18.

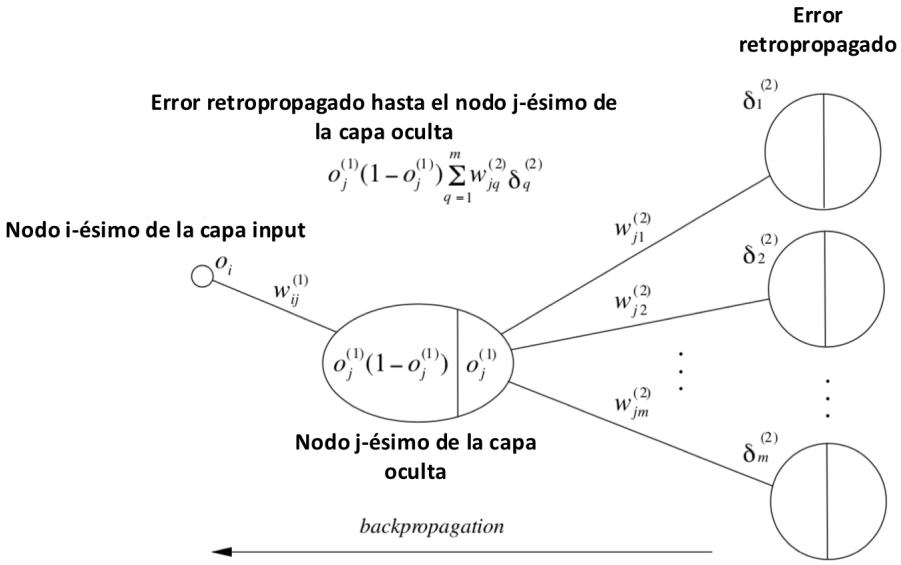


Figura 3.25: Todos los caminos de la red hasta el nodo de entrada i-ésimo.

El error retropropagado es entonces:

$$\delta_j^{(1)} = o_j^{(1)} * (1 - o_j^{(1)}) * \sum_{q=1}^{m} w_{jq}^{(2)} \delta_q^{(2)} \tag{3.20}$$

Así, la derivada parcial que estamos buscando es:

$$\frac{\partial E}{\partial w_{ij}^{(1)}} = \delta_q^{(1)} o_i \tag{3.21}$$

El error de retropropagación puede ser calculado de manera análoga a como se muestra en la imagen 3.25 para cualquier número de capas ocultas. Además las derivadas parciales de la función E mantienen la misma forma analítica.

3.7.11.1.6. Cuarto paso: actualización de pesos Después de calcular todas las derivadas parciales se procede a actualizar los pesos en la dirección negativa del

gradiente. Una constante γ define la magnitud de la corrección.

Usando las ecuaciones 3.11, 3.12 y 3.19, la regla para actualizar los pesos de las aristas que conectan la capa oculta con la capa de salida es:

$$\Delta w_{ij}^{(2)} = -\gamma o_i^{(1)} \delta_j^{(2)} \quad para \quad i \in \{1, ..., k, k+1\}; j \in \{1, ..., m\} \tag{3.22}$$

Utilizando las ecuaciones 3.11, 3.12 y 3.21 para actualizar los pesos de las artistas que conectan la capa *input* con la capa oculta obtenemos siguiente regla:

$$\Delta w_{ij}^{(1)} = -\gamma o_i \delta_j^{(1)} \quad para \quad i \in \{1, ..., n, n+1\}; j \in \{1, ..., k\} \tag{3.23}$$

Cabe recordar que usamos la convención de que $o_{n+1} = o_{k+1}^{(1)} = 1$ como habíamos indicado en 3.7.11.1.1.

Es importante hacer las correcciones de los pesos **después** de que **todos** los errores retropropagados $\delta_j^{(i)}$ con $i \in \{1, 2\}$ han sido calculados.

3.7.11.1.7. Forma matricial del algoritmo de retropropagación Cuando presentamos una red con estructura de capas podemos aplicar en el algoritmo las herramientas de álgebra lineal de una manera más sencilla. Esto nos ayuda describir los cálculos en la práctica.

En las secciones anteriores hemos mostrado una red con una capa de entrada, una capa oculta y una capa de salida (de n, k y m nodos respectivamente). La entrada de la red representada con el vector o produce la salida $o^{(2)} = s(\widehat{o}^{(1)}\overline{W}_2)$ donde $o^{(1)} = s(\widehat{o}\overline{W}_1)$. Durante los pasos de retropropagación especificados en 3.7.11.1.2, solo necesitamos los primeros n renglones de la matriz $\overline{W}_1$. Denotaremos a esta matriz de dimensión n x k como W_1. De la misma manera, la matriz de dimensión k x m la denotaremos W_2 y está compuesta por los primeros k renglones de la matriz $\overline{W}_2$. Hacemos esta reducción porque no necesitamos retropropagar valores a los *inputs* constantes correspondientes a cada valor umbral de un nodo o neurona.[7]

Siendo k el número de nodos de la capa oculta, los valores almacenados en la parte izquierda de los nodos de la capa *output* se pueden guardar en una matriz diagonal:

[7]Hay que recordar que hicimos un pequeño truco algebraico para introducir el valor umbral de cada neurona o nodo en la red extendida como se vio en 3.7.6.

$$D_2 = \begin{bmatrix} o_1^{(2)}(1-o_1^{(2)}) & 0 & 0 & \dots & 0 \\ 0 & o_2^{(2)}(1-o_2^{(2)}) & 0 & \dots & \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & o_m^{(2)}(1-o_m^{(2)}) \end{bmatrix}$$

De la misma manera es posible guardar los valores de la parte izquierda de los nodos de la primera capa en la siguiente matriz diagonal:

$$D_1 = \begin{bmatrix} o_1^{(1)}(1-o_1^{(1)}) & 0 & 0 & \dots & 0 \\ 0 & o_2^{(1)}(1-o_2^{(1)}) & 0 & \dots & \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & o_k^{(1)}(1-o_k^{(1)}) \end{bmatrix}$$

Si definimos el vector e cuyas entradas son las derivadas de las desviaciones cuadráticas como:

$$e = \begin{bmatrix} (o_1^{(2)} - t_1) \\ (o_2^{(2)} - t_2) \\ \vdots \\ (o_m^{(2)} - t_m) \end{bmatrix} \tag{3.24}$$

entonces el vector de m entradas del error retropropagado hasta los nodos de salida está dado por la expresión

$$\delta^{(2)} = D_2 e. \tag{3.25}$$

El vector de k entradas del error retropropagado hasta la capa oculta es:

$$\delta^{(1)} = D_1 W_2 \delta^{(2)}. \tag{3.26}$$

Finalmente, las matrices W_1 y W_2 son corregidas usando las expresiones:

$$\Delta W_2 = -\gamma \widehat{o}^1 \delta^{(2)} \tag{3.27}$$

y

$$\Delta W_1 = -\gamma \widehat{o} \delta^{(1)} \tag{3.28}$$

Es necesario observar que ΔW_1 y ΔW_2 son matrices de dimensiones $n \times k$ y $k \times m$ respectivamente. Esto sucede porque en las dos ecuaciones anteriores γ es un número real y posteriormente ocurre una multiplicación matricial de los vectores. Colocamos el vector $\widehat{o}^1$ de tal manera que sea una matriz de dimensión $1 \times m$, el vector $\widehat{o}$ una matriz de dimensión $1 \times n$, el vector $\delta^{(2)}$ una matriz de dimensión $1 \times m$ y el vector $\delta^{(1)}$ una matriz de dimensión $1 \times k$.

3.8. Mejoras en el proceso de aprendizaje de un perceptrón multicapa

Al modelo de red neuronal y algoritmo de aprendizaje antes descrito se pueden aplicar diversas mejoras para que la red generalice mejor, aprenda más rápido y en general optimice su desempeño. Una discusión más amplia sobre algunas técnicas para mejorar el aprendizaje puede consultarse en [21]. Aquí se abordarán algunas brevemente.

Podemos empezar con un problema asociado al uso de la función de error cuadrático en la capa de salida de una red neuronal. Para visualizar el inconveniente consideremos una sola neurona con una sola entrada como se muestra en la figura 3.26:

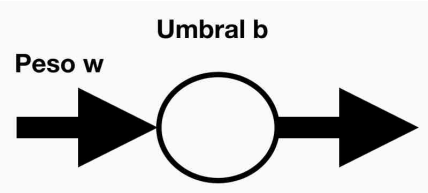


Figura 3.26: Neurona con un solo dato de entrada, un peso y un umbral.

Esta neurona recibirá una entrada con un valor de 1 y trataremos que su salida sea 0. Encontrar valores para w y b que satisfagan esas condiciones es muy sencillo y no requiere utilizar un algoritmo como el de retropropagación. Al emplear el algoritmo de retropropagación 300 veces, un factor de aprendizaje de 0.15 y empezando con valores de $w = 0.6$ y $b = 0.9$ se obtiene la gráfica que se muestra en la imagen 3.27 .

Con el entrenamiento se alcanzó una salida de 0.09. Si bien dicha cantidad no es el valor 0 que buscamos, podemos decir que es un valor muy cercano.

Pero si cambiamos ambos valores iniciales de w y b a 2.0 obtenemos la gráfica que se muestra en 3.28 .

En la figura 3.28 notamos que el costo (o error) varía poco durante la primera

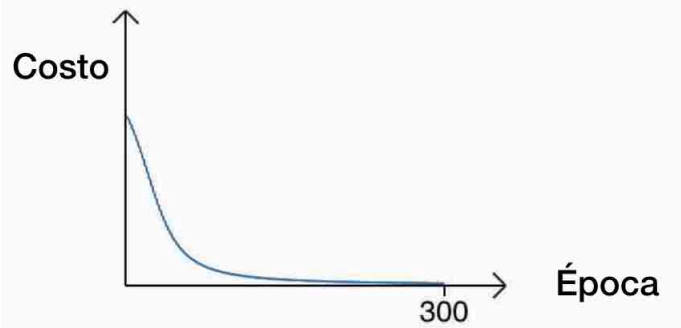


Figura 3.27: Entrenamiento de una sola neurona con un factor de aprendizaje de 0.15 durante 300 épocas. Se alcanzó una salida de 0.09

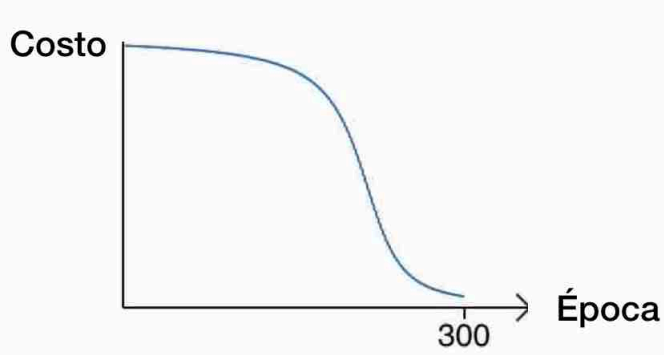


Figura 3.28: Entrenamiento de una sola neurona con un factor de aprendizaje de 0.15 durante 300 épocas. Se alcanzó una salida 0.20.

mitad de las 300 épocas. En realidad los valores de w y b apenas cambiaron durante esas épocas.

El experimento que se muestra en la figura 3.28, en las primeras épocas no hay un cambio significativo de la función costo aun cuando aplicamos el algoritmo de retropropagación (el cual modifica el valor del peso w y el del umbral b). A continuación se muestra una posible causa.

3.9. El problema de la función error cuadrático

Según hemos visto en 3.7.11.1.4 o en 3.7.10.1, el cambio del peso de una arista durante el algoritmo de retropropagación es proporcional al valor de $\frac{\partial E}{\partial w}$ y $\frac{\partial E}{\partial b}$, por lo que decir que el aprendizaje ha sido lento es lo mismo que decir que el valor de estas derivadas parciales es pequeño.

La afirmación anterior se puede comprobar con la neurona de la imagen 3.29 usando como función de costo el error cuadrático. Supongamos que el error está dado

por $E = \frac{(y-a)^2}{2}$, que $y = 0$ es la salida buscada y que a es la salida de la red cuando recibe $x = 1$.

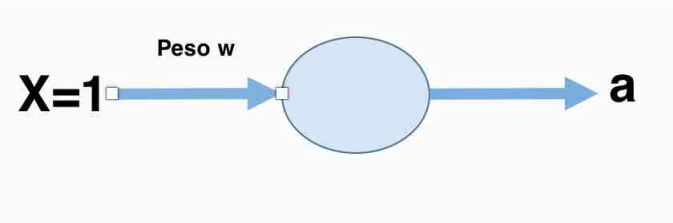


Figura 3.29: $y = 0$ es la salida buscada, a es la salida de la red cuando recibe $x = 1$.

Además sabemos que la salida de la red a se calcula con la función sigmoide que denotamos como σ, es decir $a = \sigma(z)$ donde $z = w * x + b$. Aplicando la regla de la cadena para obtener las parciales $\frac{\partial E}{\partial w}$ y $\frac{\partial E}{\partial b}$ se tiene que:

$$\frac{\partial E}{\partial w} = (a - y) * \sigma'(z) * x = a * \sigma'(z) * x \tag{3.29}$$

$$\frac{\partial E}{\partial b} = (a - y) * \sigma'(z) = a * \sigma'(z) \tag{3.30}$$

En las ecuaciones anteriores se ha sustituido el valor de $x = 1$ y $y = 0$.

Como podemos ver en la imagen 3.31, la forma de la función σ es muy plana en algunas partes, es decir, la pendiente de la recta tangente en esos puntos es casi horizontal. De tal manera que σ' debe ser un valor cercano a cero para valores de z muy grandes.

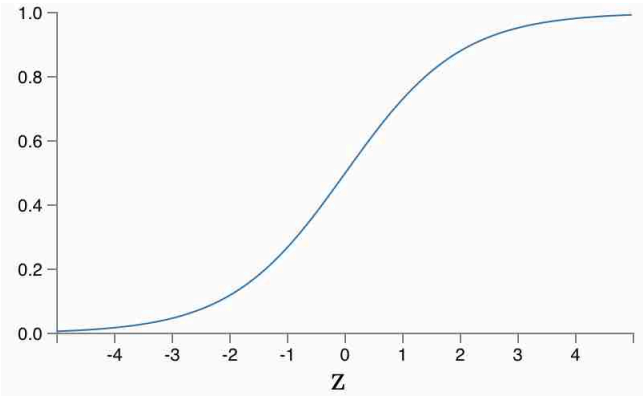


Figura 3.30: Función sigmoide.

Para acelerar la rapidez del aprendizaje podríamos cambiar la función de error cuadrático que habíamos empleado.

3.10. Función de entropía cruzada

Reemplazamos la función del error cuadrático por una función conocida como función de entropía cruzada o *cross-entropy cost function*. Para entender mejor esta función supongamos que tenemos la red que se muestra en la figura 3.31. Esta red consta de una sola neurona y deseamos una salida que denotaremos por $y \geq 0$.

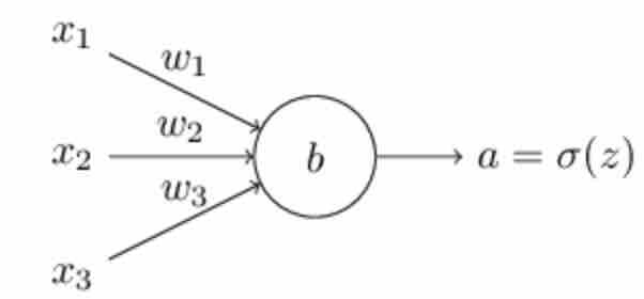


Figura 3.31: Red neuronal de una sola neurona con entradas $x_1, ..., x_3$ y *output* $a = \sigma(z)$ con $z = \sum_{i=1}^{3} w_i * x_i + b$ la suma ponderada de las entradas.

La función de entropía cruzada la definimos como:

$$C = -\frac{1}{n} \sum_{x} [y * ln(a) + (1 - y) * ln(1 - a)] \tag{3.31}$$

En donde n es el número de ejemplos de la base de entrenamiento, la suma es sobre todos los ejemplos de entrenamiento $x^i = (x_1^i, x_2^i, x_3^i)$ y $y \in \{0, 1\}$ es la salida deseada para x^i. [8]

Tenemos que cerciorarnos que la función de entropía cruzada es en efecto una función costo. Para ello notemos que:

1. $C > 0$ pues todos los términos en la suma que definen a la función C son negativos ($a = \sigma(z) > 0$ una sigmoide cuyo valor se encuentra en el intervalo $(0, 1)$ y $y \geq 0$) y al final de la suma encontramos un signo menos.

2. Si la salida de la red está cerca de la salida deseada para cada entrada x entonces $C \approx 0$. Por ejemplo, supongamos que la salida deseada para cierto ejemplo x es $y = 0$ y el valor arrojado por la red es $a \approx 0$, entonces $y * ln(a) \approx 0$ y $(1 - y) * ln(1 - a) \approx 0$.[9]

[8] Notemos que n y el número de índices sobre los que corre la suma no tienen porque ser iguales ya que podríamos usar pocas entradas si usamos el descenso del gradiente estocástico.

[9] Aplicamos $\lim_{a \to 0} C(a)$. Además se aplican algunos propiedades de la función $ln(x)$.

La función de entropía cruzada esquiva el problema mencionado en la parte 3.9. Para comprobarlo podemos obtener las parciales de la función de entropía cruzada con respecto de sus pesos. Además sustituimos $a = \sigma(z)$ en la definición de la función de entropía cruzada y aplicamos la regla de la cadena.

$$\frac{\partial C}{\partial w_j} = -1 * \frac{1}{n} * \sum_x [(\frac{y}{\sigma(z)} - \frac{(1-y)}{(1-\sigma(z))})] * \frac{\partial \sigma}{\partial w_j} \tag{3.32}$$

Sustituyendo $\frac{\partial \sigma}{\partial w_j} = \sigma'(z) * x_j$ se tiene que:

$$\frac{\partial C}{\partial w_j} = -1 * \frac{1}{n} * \sum_x [(\frac{y}{\sigma(z)} - \frac{(1-y)}{(1-\sigma(z))})] * \sigma'(z) * x_j \tag{3.33}$$

Simplificando:

$$\frac{\partial C}{\partial w_j} = -1 * \frac{1}{n} * [\sum_x \frac{\sigma'(z) * x_j}{(1-\sigma(z)) * \sigma(z)} * (\sigma(z) - y)] \tag{3.34}$$

Usando la definición de la función sigmoide $\sigma(z) = \frac{1}{(1+e^{-z})}$ y obteniendo la derivada de la función sigmoide tenemos que $\sigma'(z) = (1-\sigma(z)) * \sigma(z)$. Por lo que al sustituir en la ecuación anterior este resultado se sigue que:

$$\frac{\partial C}{\partial w_j} = -1 * \frac{1}{n} * \sum_x x_j * (\sigma(z) - y) \tag{3.35}$$

Esta última expresión nos muestra que el aprendizaje depende de la expresión $(\sigma(z) - y)$; es decir, la intensidad del aprendizaje viene dado por la magnitud del error que cometió la red. Esto evita que el aprendizaje este relacionado con la expresión $\sigma\prime(z)$ que causaba problemas en la función error cuadrático.

Nuestra definición de función de entropía cruzada es aplicable para el ejemplo sencillo de red que introducimos en la imagen 3.31, pero es posible generalizarla para una red estructurada por capas. Si $y_1, ..., y_n$ son los valores deseados en la capa de salida y $a_1^L, ..., a_n^L$ los valores de salida reales de la red, m el número de ejemplos de entrenamiento, la función de entropía cruzada se define como:

$$C = -1 * \frac{1}{m} \sum_x \sum_j [y_j * ln(a_j^L) + (1-y_j) * ln(1-a_j^L)] \tag{3.36}$$

La suma $\sum_j$ es sobre todas las neuronas de la capa de salida.

Al igual que en el caso simple, se puede comprobar que esta función resuelve el problema descrito en la parte 3.9.

3.11. Función Softmax

Haremos uso de una función de activación diferente a la función sigmoide en las neuronas de la capa de salida. Esta función se le conoce como Softmax y se define como [21]:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{3.37}$$

Con $z_j^L = \sum_k [w_{jk}^L a_k^{L-1} + b_J^L]$ la entrada ponderada que ingresa a la neurona de la capa de salida con índice j. La suma del denominador en la ecuación 3.37 corre sobre todas las neuronas de la capa de salida. Hay que recordar el esquema mostrado en la imagen 3.32.

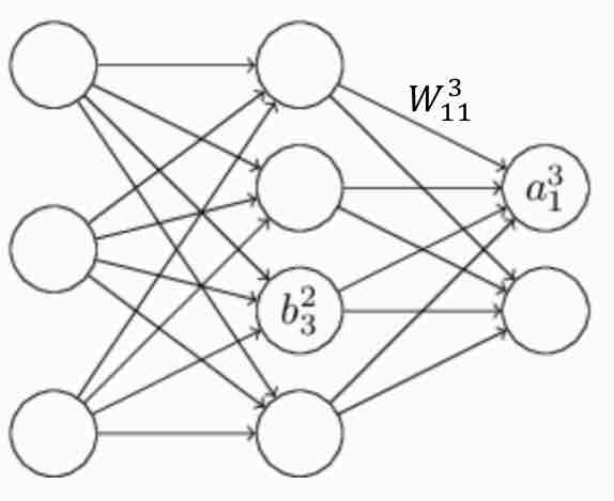


Figura 3.32: En este ejemplo $z_1^3 = \sum_k [w_{1k}^3 a_k^2 + b_1^3]$ donde se suma sobre todas las neuronas de la segunda capa.

La primer propiedad que observamos sobre esta función es que la suma de las salidas, en la capa de salida, es 1. De allí que el vector de salida podría ser considerado como una especie de vector de probabilidad. Es decir, siendo L la capa de salida de la red, se cumple que:

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1 \tag{3.38}$$

En el problema que concierne a esta tesis, la función Softmax nos ayuda a que la salida quede expresada como "la alarma que es más probable que ocurra". La función de error cuadrático medio no necesariamente presenta esa propiedad.

La función Softmax también nos ayuda a esquivar el inconveniente de un aprendizaje lento mencionado en 3.9 con algunas modificaciones a la función costo. Supongamos que tenemos una entrada de entrenamiento que denotaremos por x y una sola salida deseada y para esa entrada. Definimos la función de verosimilitud (o *log-likelihood function* en inglés) para a_y^L como:

$$C = -ln(a_y^L) \tag{3.39}$$

Esta función resulta ser una función costo. Para comprobarlo supongamos la salida deseada es y y que la red ha hecho bien su trabajo, entonces tendremos que el valor a_y^L debería ser cercano a 1. De aquí que $-ln(a_y^L) \approx 0$. Por el contrario, si $a_y^L \approx 0$ (la red no cree que la entrada x deba clasificarse como y), entonces el valor de $-ln(a_y^L) >> 0$ es muy grande (hay un signo menos antecediendo a la función logaritmo).

Si calculamos las parciales en este ejemplo tenemos que:

$$\frac{\partial C}{\partial w_{jk}^L} = (a_k^L)^{-1} * (a_j^L - y_j) \tag{3.40}$$

$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j \tag{3.41}$$

Con b_j^L el valor umbral de la neurona j-ésima de la capa de salida L, la cual podía ser vista como una arista de la red según hemos visto anteriormente.

Las derivadas parciales de la función C son similares a las que obtuvimos para la función costo de la entropía cruzada. La mejora a los pesos se ve directamente influida por el factor $(a_j^L - y_j)$.

3.12. *Overfitting* o sobreajuste

Un problema que se presenta en el área del aprendizaje automático es el sobreajuste. De presentarse, el modelo se vuelve inútil para predecir o pronosticar nuevos datos (datos que no fueron usados durante el entrenamiento) puesto que hemos hecho todo lo posible por ajustar el modelo a datos particulares (consúltese [4]) .

Para ejemplificar mejor esta situación, supongamos que los datos son divididos en dos partes: datos entrenamiento y datos de prueba. Estos datos de prueba **no serán utilizados para el entrenamiento de red**. Nos servirán para evaluar qué tan bien se desempeña.[10]

Figura 3.33: División de los datos disponibles.

Supongamos además que contamos con una función costo y que procedemos a entrenar la red.

Al terminar el entrenamiento, con los datos destinados para dicho propósito, graficamos el error producido en cada época o *epoch* obteniendo la gráfica que se muestra en la figura 3.34.

Sin fijar nuestra atención en multitud de detalles y parámetros que puede tener la red o su arquitectura, podríamos pensar que su desempeño va mejorando cada vez más. Pero al graficar el porcentaje de ejemplos que la red clasifica correctamente (usando los datos para *test* o prueba) se obtiene la gráfica que se muestra en la figura 3.35.

También podríamos analizar el error producido con los datos que están destinados para *test* o prueba al mismo tiempo que realizamos el entrenamiento (con los ejemplos de entrenamiento por supuesto). Supongamos que dicho historial muestra la gráfica mostrada en la imagen 3.36.

En la gráfica 3.35 se observa que el entrenamiento no está mejorando la clasificación para los ejemplos destinados a *test*. En la época 280, aproximadamente,

[10]Cada ejemplo posee una entrada y su salida deseada.

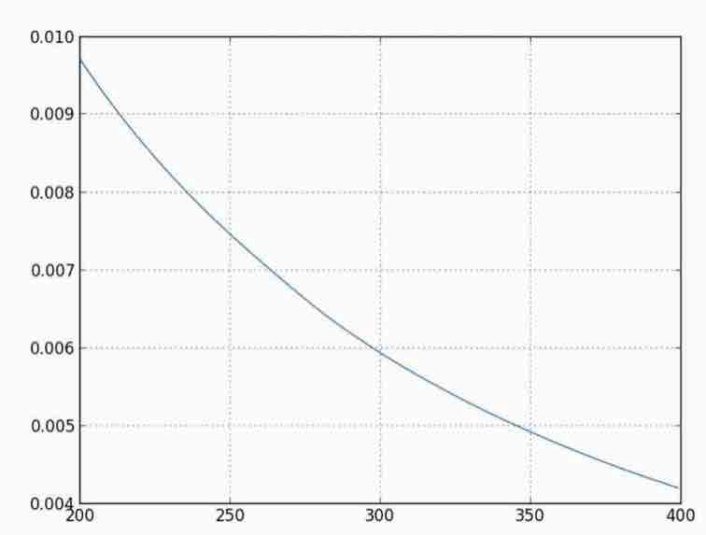


Figura 3.34: Gráfica de la función error con respecto de las épocas

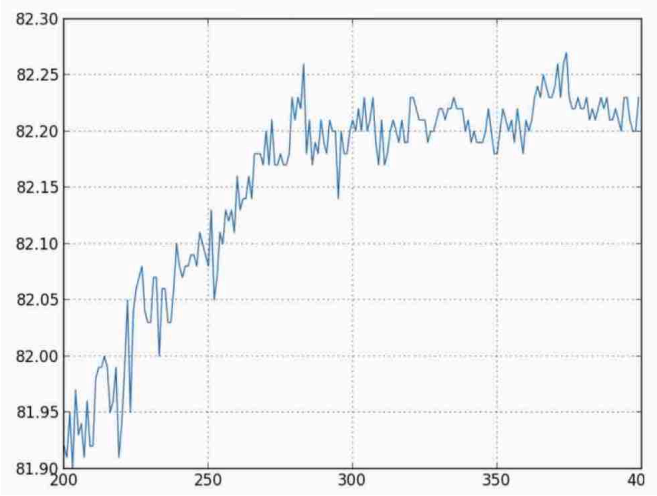


Figura 3.35: Gráfica del porcentaje de los ejemplos destinados a *test* clasificados correctamente al transcurrir las épocas.

se alcanza un porcentaje que posiblemente sea alrededor del cual oscilan los siguientes porcentajes. Decimos que la red está sobreajustada si continuamos entrenando pues no mejora la clasificación a pesar de que el error que nos arroja la función costo en los datos de entrenamiento es cada vez menor.

De manera intuitiva podría decirse que la red se está aprendiendo particularidades de los datos de entrenamiento pero está perdiendo capacidad para generalizar o incluso interpolar valores desconocidos (los datos que reservamos para probarla o de *test*). Dicha situación no es deseable, puesto que en realidad desconocemos algunos datos y sus verdaderos valores .

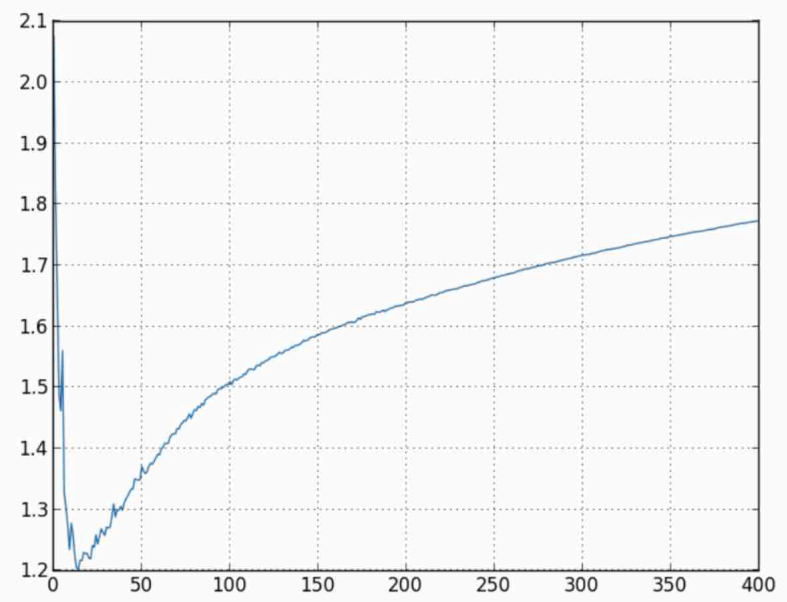


Figura 3.36: Gráfica del error cometido en los datos de *test* o prueba al cambiar las épocas.

Para evitar la situación anterior podríamos parar el entrenamiento cuando el porcentaje de aciertos en el conjunto de prueba no mejore o el error vuelva a ser considerablemente alto.

Emplearemos un porcentaje de los datos destinados al entrenamiento para detectar si nuestra red se está sobreajustando. Es decir, dividiremos los datos originales como en el esquema que se muestra en la imagen 3.37.

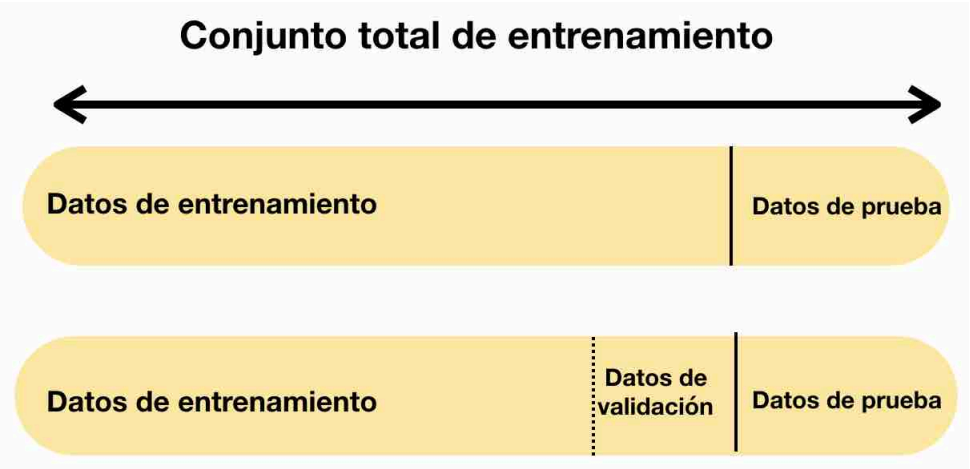


Figura 3.37: División de los datos disponibles. Se destina un porcentaje, generalmente del 20 %, de los datos de entrenamiento como datos de validación.

La manera tradicional de detectar el sobreajuste es analizar el error en los

datos de entrenamiento y en los datos de validación. Con esto los datos de test podrían servir para otros fines, por ejemplo evaluar que tan buenos son otros parámetros como el factor de aprendizaje usado en el algoritmo de retropropagación.

3.13. Evaluación del desempeño de una red neuronal

En esta sección se discute brevemente algunas maneras de evaluar el desempeño de la red neuronal y algunos conceptos relacionados.

3.13.1. Muestreo

El muestreo estadístico es un factor decisivo para el buen desempeño del modelo. Una vez que hemos entrenado la red neuronal podemos emplear algunos ejemplos para evaluar el modelo. Si bien podemos emplear los mismos ejemplos que usamos para entrenamiento, lo ideal es someter a la red neuronal a situaciones nuevas y corroborar su capacidad de generalización.

Nuestro modelo funcionará mejor si entrenamos a la red con ejemplos suficientes y representativos de todo el conjunto de ejemplos. De igual manera, para evaluar el desempeño de la red neuronal precisamos de un conjunto de ejemplos que sean representativos y en buena cantidad.

En la imagen 3.37 hemos hecho una partición de todos los ejemplos disponibles con el objetivo de disminuir el sobreajuste durante el entrenamiento; sin embargo, es necesario que los ejemplos destinados a formar tanto el conjunto de entrenamiento como el conjunto de prueba o *test* **sean conjuntos cuyas propiedades sean extrapolables a todos los ejemplos de los que disponemos originalmente**, razón por la cual se aplican diversas técnicas de **muestreo estadístico**. Estas técnicas contemplan la naturaleza de los datos, su dispersión, sus interrelaciones, la cantidad de ellos disponibles [11] y otras características para seleccionar ejemplos de ambos conjuntos; particularmente el conjunto de ejemplos de prueba nos brindará una mejor perspectiva de lo que es capaz de hacer la red neuronal bajo ciertos parámetros. Construir un modelo de aprendizaje automático -como la red neuronal- parsimonioso y útil en la práctica puede depender de qué tan bien hemos llevado a cabo la labor de muestreo.

Existen diversas técnicas de muestreo estadístico. Algunas de ellas son el

[11] La cantidad de ejemplos en ocasiones no suele ser relevante. Un ejemplo de esto puede ser una imagen con una resolución de 1200 x 1800 a la cual solo podemos acceder mediante un muestreo de los píxeles que la conforman. Podemos elegir 15,000 píxeles de los bordes o 1,000 píxeles uniformemente distribuidos. El segundo conjunto nos proporciona más información sobre el contenido de la imagen. La idea es obtener una muestra que englobe la mayor varianza de la población o datos a estudiar.

muestreo aleatorio simple, muestreo estratificado o el muestreo por conglomerados. Es posible hablar de muestreo no probabilístico como es el caso del muestreo por cuotas. Para un estudio más detallado del muestreo y sus técnicas puede consultarse [3].

Subajuste o *underfitting*. Una mala muestra puede desembocar en el fenómeno contrario al sobreajuste, conocido como subajuste o *underfitting*. Un modelo que incurre en el subajuste es demasiado sencillo con respecto de los datos y el problema con el que se está trabajando(véase [4] para una exposición más amplia) . Esta situación se muestra en la imagen 3.38 para un modelo de regresión (ajuste de una línea a los datos). Si la red neuronal se entrena con pocos ejemplos o los ejemplos proporcionados no son representativos, tendrá poca capacidad de generalización.

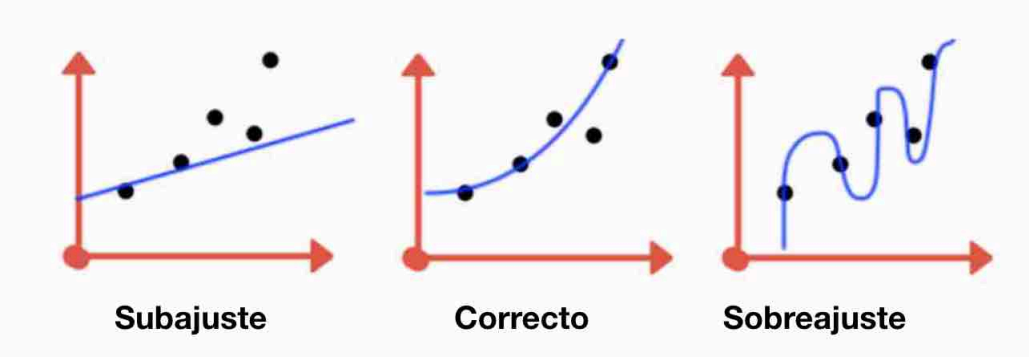


Figura 3.38: Subajuste (*underfitting*), ajuste considerado adecuado y sobreajuste (*overfitting*).

3.13.2. Matriz de confusión

En un modelo de clasificación cualquiera es posible obtener una precisión muy alta en una clase, pero muy mal en otras. Si además se presenta la situación en la que la clase muy bien clasificada es la que tiene mayor presencia (está sobrerepresentada), la precisión global puede ser engañosa. La precisión que hemos manejado hasta el momento no nos muestra si todas las clases son bien predichas o solo un subconjunto de ellas.

La matriz de confusión nos ofrece un mejor detalle de las predicciones logradas por la red neuronal pues desglosa la manera en que ésta se confunde o acierta. En la matriz de confusión cada columna representa el número de predicciones en cada clase, mientras que cada renglón representa a las instancias en la clase real.

Su valía consiste en que no solo muestra los errores que comete la red neuronal - algo que ya nos mostraba la precisión- sino que incluso nos permite obtener los tipos de error definidos en la rama de la estadística conocidos como errores del tipo 1 y tipo 2 (véase [14]).

Para construir una matriz de confusión se necesita:

1. Ejemplos de prueba o *test* (o validación) con etiquetas que nos indiquen cuál es el valor esperado para cada ejemplo.
2. Hacer una predicción con la red neuronal sobre cada uno de los ejemplos que tenemos en el conjunto de prueba.
3. De las etiquetas esperadas y las etiquetas predichas hay que contar:
   - La cantidad de predicciones correctas hechas en cada clase.
   - La cantidad de predicciones incorrectas hechas en cada clase, organizadas por la clase que fue predicha.

Como ya se había mencionado, estos valores se colocan en una matriz en la cual:

- cada renglón representa a las instancias en la clase real.
- cada columna de la matriz representa el número de predicciones de cada clase.

Los conteos de ejemplos correctamente clasificados e incorrectamente clasificados se colocan en esa matriz. En ocasiones se intercambian los renglones por las columnas, lo que resulta en que cada renglón representa el número de predicciones de cada clase y cada columna representa las instancias en la clase real.

		Valor Predicho		
		Gato	**Perro**	**Conejo**
Valor Real	**Gato**	5	3	0
	Perro	2	3	1
	Conejo	0	2	11

Figura 3.39: Ejemplo de una matriz de confusión.

Un ejemplo de matriz de confusión con más de tres clases se muestra en la sección de experimentos 5.28.

A partir de la matriz de confusión surgen algunos criterios utilizados para la evaluación del desempeño de la red neuronal. Dado el conjunto de prueba y para una clasificación binaria podemos clasificar instancias (ejemplos) que pertenecen a la clase **A** y los que no. Algunos de los criterios que se emplean comúnmente se muestran a continuación.

- *Classification Accuracy (CA)*: es la proporción de instancias clasificadas correctamente del total de instancias [12] del conjunto de prueba. Coincide con ser la suma de la diagonal de la matriz de confusión entre el número de instancias en el conjunto de prueba.

- *Precision*: Es la proporción de ejemplos de la clase **A** clasificados correctamente entre el total de las instancias que la red neuronal propone como clase **A**.

- *Recall*: Es la proporción de las instancias que la red clasifica pertenecientes a la clase **A** entre todas las instancias de la clase **A**. Alcanza su mejor valor en 1 (cuando *precision* y *recall* valen ambos 1) y su peor valor en 0 (cuando ambos criterios valen 0).

- *F-1 score* o F-1 valor: es la media armónica entre *precision* y *recall*.

- Área debajo de la curva o *Area Under the Curve* (AUC): Es el área debajo de una curva que surge como función de dos criterios conocidos como sensibililidad (*sensitivity*) y especifidad(*specificity)*. Estos últimos criterios son calculados vía la matriz de confusión. Una exposición más detalla puede encontrarse en [11].

Si bien estos criterios se suelen elaborar a través del conjunto de prueba o *test*, pueden obtenerse utilizando un conjunto cualquiera como lo podría ser el mismo conjunto de entrenamiento.

Por otro lado, el análisis de una clasificación que no sea binaria se pude llevar a cabo con estos criterios de la siguiente manera: para cada clase i se elabora la matriz de confusión que divide las instancias en los que pertenecen a la clase i y los que no pertenecen a la clase i, calculándose los criterios antes mencionados. De esta manera tenemos el criterio CA, por ejemplo, para la clase 1, el CA para la clase 2 y así sucesivamente. Si buscamos un criterio CA para el modelo en general podemos promediar los CA de cada clase.

En la literatura se señala que decantarse por uno u otro criterio suele depender de diversas consideraciones, como puede ser el entendimiento de la parte conceptual del modelo e incluso, en ocasiones, la pericia de la persona que esté haciendo el análisis. Además, es posible que para ciertos modelos necesitemos de criterios personalizados.

En los experimentos realizados en este texto solo haremos referencia a los criterios anteriormente listados.

[12] O ejemplos de la red.

Capítulo 4

Descubrimiento de patrones en secuencias de alarmas con redes neuronales

4.1. Alarmas

Como ya se ha comentado a lo largo de esta tesis, usaremos una red neuronal para detectar patrones en secuencias de alarmas con el objetivo de predecir fallas.

Los datos utilizados en los experimentos proviene de un conjunto de alarmas reales que corresponden a tres meses de operación de una subred de telecomunicaciones. Estas alarmas en realidad corresponden al fallo de 7 elementos de red. Nuestros eventos serán los elementos de red que fallan y no las alarmas propiamente dichas, las cuales poseen más información que simplemente el fallo de un elemento de red. Adicionalmente a los datos reales, para probar los alcances del método se diseñó un serie de experimentos de control que nos permitirán evaluar el desempeño de la red neuronal en escenarios que se han descrito en la literatura con datos sintéticos.

Como ya se había establecido, usaremos sin distinción los términos eventos y alarmas a lo largo de este capítulo.

En virtud de lo anterior, a cada elemento de red podemos asociarle un número entero a manera de *etiqueta*, digamos que $A \to 1, B \to 2, ...$, lo que nos proporciona una biyección entre los eventos y los primeros k naturales. Es muy importante notar que dichas alarmas son variables categóricas que no son comparables y podrían ser representadas con letras, colores, etc. Por lo que biyectar el conjunto de alarmas con un conjunto cualquiera, cuyos elementos no tienen un orden, es válido.

Dado que el tiempo se considera discreto en esta tesis, se asumirá que en

un momento determinado solo se recibe el fallo de un elemento de red. Las alarmas se ordenan según el orden en que hayan sido recibidas. Es decir, recibimos la lista ordenada de las *alarmas* con un vector de n entradas $X = (x_1, ...x_n)$. Dado que una ventana es un fragmento de la secuencia de eventos original, dicha ventana podemos representarla como un vector $V \in \mathbb{N}^n$ (donde n es el tamaño de ventana). Por ejemplo, podríamos considerar una lista de alarmas como $X = \{4, 1, 2, 5, 7, 7, 7, 7, 1, 2, 3, 3\}$.

Con los criterios antes mencionados, se construyó una secuencia de alarmas que consta de 287,256 elementos, la cual será usada para poner a prueba la red neuronal.

La totalidad de las alarmas se distribuye como se muestra en la figura 4.1. La figura 4.2 muestra los porcentajes de aparición.

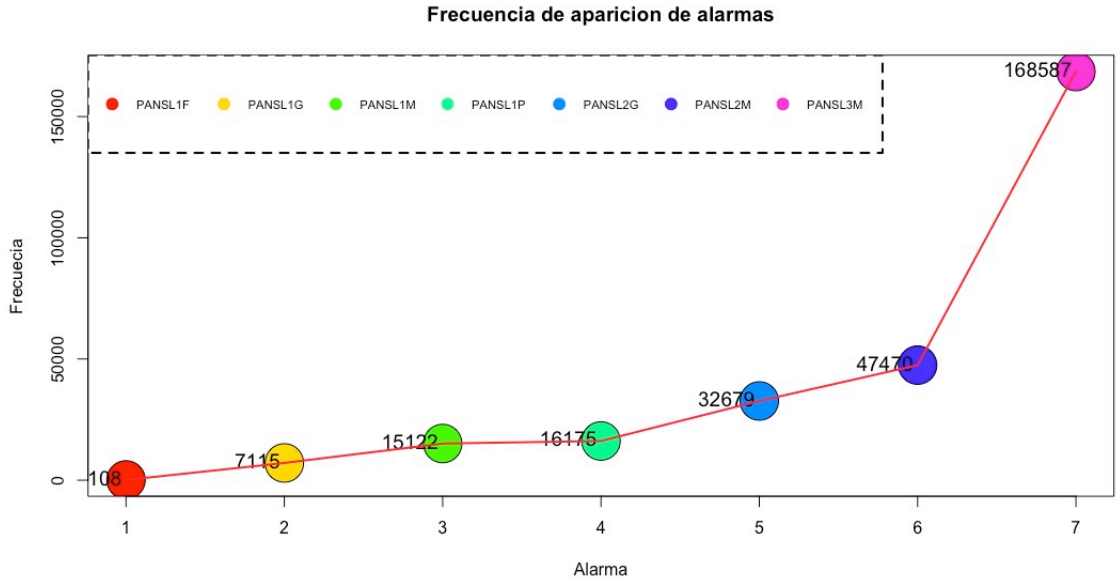


Figura 4.1: Distribución de las alarmas.

4.2. Modelado de los datos de entrada

Se puede proponer varias maneras para codificar las entradas a la red neuronal. Cada una de ellas conlleva ciertas ventajas e inconvenientes. A continuación se presentan tres maneras de transformar la secuencia de eventos en entradas y salidas para una red neuronal. Las tres alternativas consideran que el tiempo es discreto.

- *Usar el vector asociado a la ventana*: Cada vector representa una ventana de la secuencia. La entrada i-ésima de un vector corresponde al i-ésimo evento de una

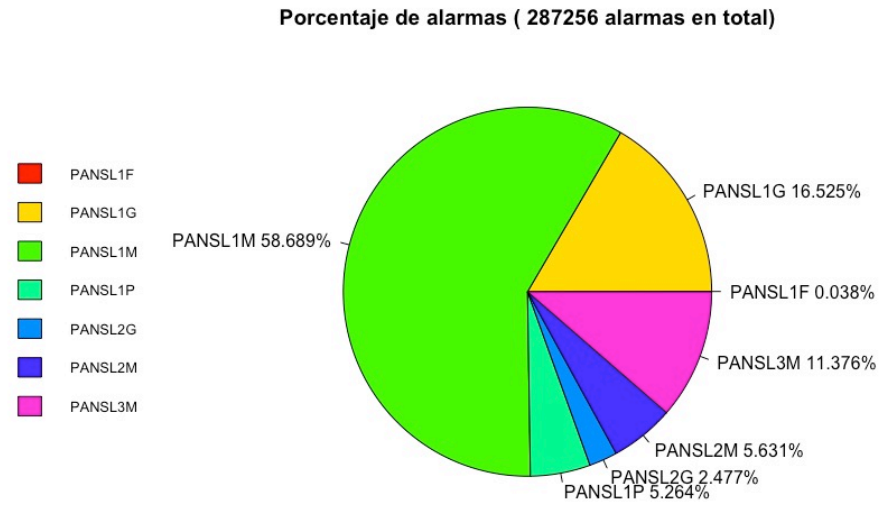


Figura 4.2: Proporción de la frecuencia de aparición de las alarmas

ventana. Reservamos $0 \in \mathbb{N}$ para representar la ausencia de una alarma en la ventana. Un ejemplo de esta representación puede ser una ventana con el vector asociado (3,1,0,8,0,0,2). Esta representación significa que se recibió la alarma 3 en la posición 1 de la ventana, la alarma 1 en la posición 2 de la ventana, ninguna alarma en la posición 3 de la ventana, etc. En esta representación se conserva el orden de los eventos pero no se contempla la frecuencia con la que se presentaron las alarmas (es decir, no se introduce la información de la frecuencia a la red neuronal).

- *Frecuencia de los eventos que sucedieron en una ventana*: Formamos un vector que tenga tantas entradas como alarmas posibles, denotemos a ese número como n. Es decir, formamos un vector $X \in \mathbb{N}^n$ en el que la entrada i-ésima de dicho vector corresponde a la frecuencia de ocurrencia de la alarma i en una ventana. En esta representación se pierde el orden de los eventos.

- *Un vector que representa una ventana y contiene la frecuencia de los eventos ocurridos en una ventana*: Una tercera opción es utilizar un vector que posea la información de los eventos ocurridos en una ventana y su frecuencia. Se construye una matriz cuya entrada (i,j) nos dice que la alarma cuya categoría es i ha sucedido j veces en la ventana. La matriz posee tantos renglones como categorías de alarmas. Dicha matriz se introduce a la red en forma de un vector, cuyas entradas resultan ser algún reordenamiento de las entradas de la matriz. En esta representación también se pierde el orden de los eventos.

4.3. Modelado de los datos de salida

La salida de la red neuronal también puede ser definida de diversas maneras. La más práctica para nuestros fines es definir la salida como un vector con las probabilidades de ocurrencia de cada evento, pero no es la única posible.

La salida de la red neuronal va a consistir en un vector de k entradas, es decir, en un vector $Y \in \mathbb{R}^k$ en el que $\forall\ i \in \{1, ..., k\}$ se tiene que $y_i \in [0, 1]$, representando y_i probabilidad de la ocurrencia de cada evento dentro de un tiempo especificado. Un ejemplo de salida podría ser $(0.002, 0.3, ..., 0)$, donde cada entrada del vector es la probabilidad de que suceda una alarma en n unidades de tiempo después de la ocurrencia de la última alarma de la ventana. En esta tesis se aborda el caso de $n = 1$. La entrada de valor más alto de este vector nos indicaría la alarma más probable.

En nuestro caso usaremos un vector de 7 entradas como salida. Eso significa que utilizaremos 7 neuronas de salida cuya función de activación es la función conocida como "Softmax".

4.4. Matriz de entrenamiento

Necesitamos un conjunto de ejemplos de entrenamiento para entrenar a la red neuronal. Cada pareja de vectores entrada-salida definida en las secciones 4.2 y 4.3 constituye un ejemplo del conjunto de entrenamiento o del conjunto de validación. Para generarlo, debemos deslizar la ventana a lo largo de la secuencia. El tamaño del paso puede producir diferentes conjuntos. En este trabajo la ventana se fue deslizando una posición, es decir, el paso fue de tamaño uno.

Como vector de entrada usaremos el primer enfoque mencionado en 4.2, es decir, fijaremos el tamaño de ventana en n. Para ello crearemos ventanas sobre la lista de alarmas a semejanza de lo que vemos en la figura 2.2 del capítulo 2.5 y las guardamos como un vector de dimensión n. Dada una ventana (de tamaño n), nuestro objetivo será predecir la alarma que se encuentra en la posición n a partir de las $n-1$ alarmas anteriores que se observaron.

Nuestra red recibirá un vector de dimensión $n-1$ y a diferencia de lo que se muestra en la imagen, nuestra primer ventana empezará desde la primera alarma y termina en la alarma que se encuentra en la posición n-ésima. Si el tamaño de ventana es n y como nuestra lista consta de 287,256 alarmas, tendremos en total un total de $287,256 - n$ ventanas.

Por último, la matriz de entrenamiento tendra como renglón i-ésimo la ventana que empieza en la alarma i-ésima.

4.5. Comentarios sobre el uso de una red neuronal en el problema de la predicción de alarmas

Antes de emplear una red neuronal para la predicción de alarmas podríamos preguntarnos si la red puede llevar a cabo tal tarea, es decir, si el problema es soluble con una red neuronal. También podríamos preguntarnos si en efecto se trata de un problema de clasificación. En este trabajo no se dará una respuesta categórica a estas preguntas pero se presentan los resultados, así como ideas y razonamientos que apoyan nuestra confianza en el enfoque utilizado.

Se presentan cuestionamientos similares con otros modelos matemáticos que están relacionados con la predicción, como las series de tiempo, en los cuales solo conocemos algunos datos y valores de la función y tratamos de ajustar una función muy parecida. En cierta manera, se usan los datos del pasado (o disponibles) esperando que los datos restantes posean un comportamiento similar. Esto constituye un primer supuesto en cualquier intento de modelación.

Por otro lado, ¿podríamos hablar de causalidad al observar en reiteradas ocasiones la aparición de cinco alarmas previo a la ocurrencia de una sexta? No siempre correlación implica causalidad (véase [22]).

Si observamos la presencia de una alarma A cualquiera precediendo muchas veces una alarma B ¿podemos afirmar que la alarma A es la causante de la alarma B? Una representación de la situación se muestra con el siguiente grafo dirigido 4.3.

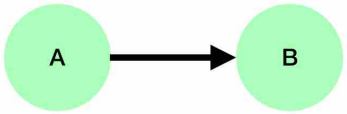


Figura 4.3: Grafo representando la alarma A como causante de la alarma B. Las aristas dirigidas representan la causalidad.

Pero podrían existir variables ocultas o factores que pudimos pasar por alto en nuestro modelo, y que pueden afectar tanto a la ocurrencia de la alarma A y la alarma B. Esta situación se representa en el siguiente grafo 4.4.

Al respecto hay diversos enfoques que abordan el problema. Una presentación amplia sobre el tema puede leerse en [22] o bien, una breve introducción también es posible leerla en [20].

En la presente tesis se espera que al usar la red neuronal se llegue a conclusiones sólidas gracias a que a priori hay relaciones entre las alarmas que son conocidas por los expertos en el mundo de las telecomunicaciones. Además el porcentaje de alarmas predichas correctamente una vez que tenemos la red neuronal puede darnos

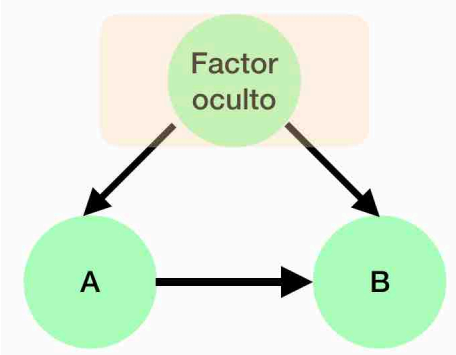


Figura 4.4: Grafo representando la causalidad de una variable oculta o factor oculto.

indicios de que la red está cumpliendo las expectativas. Este porcentaje es considerablemente alto gracias a la estructura interna con la que se presenta la lista ordenada de alarmas, lo cual no se puede afirmar de cualquier secuencia. Algunos experimentos de control del apéndice 5.5 nos muestran que el desempeño de la red neuronal puede cambiar mucho utilizando ciertas secuencias.

4.6. Arquitectura de la red neuronal

Figura 4.5: Representación del vector de alarmas, una ventana de longitud o tamaño 5 y su respectiva salida deseada o *target*.

Estructura de la red neuronal. Para diseñar la arquitectura de la red neuronal se realizaron experimentos exploratorios. Se observó que al agregar más capas ocultas, no se obtenían mejoras sustanciales a una red neuronal totalmente conectada con una sola capa oculta. La capa de entrada debe coincidir con el tamaño de la ventana que deseamos utilizar (tamaño n). En todos los experimentos las neuronas en las capas ocultas poseen una función de activación conocida como ReLU cuya definición es:

$$f(x) = x^+ = max(0, x) \tag{4.1}$$

Finalmente, la red tendrá una capa de salida con siete neuronas (las cuales coinciden con el número de alarmas posibles en la secuencia). En cada una de estas

neuronas se utiliza la función *Softmax* como función de activación, definida para la j-ésima neurona de la capa de salida (denotada por L) como:

$$a_j^L = \frac{e^{z_j^L}}{\sum_{n=1}^{7} e^{z_k^L}} \quad k \in \{1, 2, ..., 7\} \tag{4.2}$$

En resumen se utiliza una red neuronal con la siguiente estructura:

- Capas ocultas: 1
- Neuronas en la capa oculta: 15
- Factor de aprendizaje de 0.01
- Función de activación de las neuronas de la primera capa: relU [1]
- Función de activación de las neuronas de la capa de salida: Softmax
- Función error o costo utilizada: función de entropía cruzada o *categorical cross-entropy*.
- Algoritmo utilizado para el aprendizaje: Retropropagación (usando método del descenso del gradiente estocástico).
- Tamaño del *mini-batch*[2]: 30
- Cantidad de datos usados para test: Se comienzan los experimentos con el último 20 % de las ventanas de la secuencia. Posteriormente se toman ventanas al azar.
- Cantidad de datos de validación: 20 % de los datos usados para entrenamiento

El criterio de paro utilizado consiste en detener el entrenamiento si no se logra disminuir, de manera considerable, el valor de la función costo o error en los datos de validación al final de cada época. Si denotamos por E_t el valor de la función costo en los ejemplos de validación en la época t , diremos que el entrenamiento en una época n no mejoró si $E_{n-1} < E_{n+1} - 0.00001$. Se permiten 3 épocas consecutivas en las que se cumple el criterio antes de detener el entrenamiento.

[1] Véase [5] y [21] para más información sobre esta función de activación
[2] Cantidad de ejemplos usados para entrenamiento en cada época

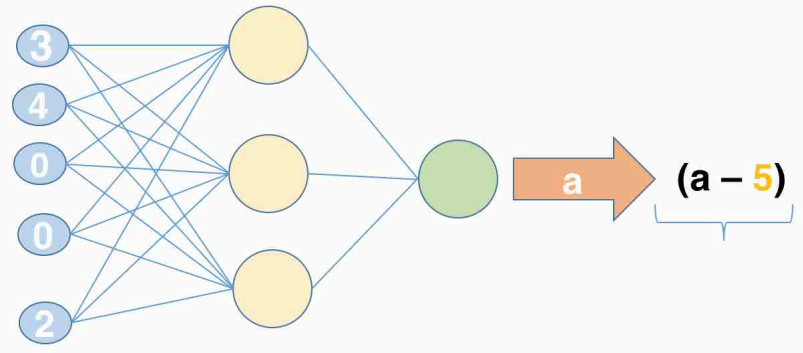


Figura 4.6: Representación de la red procesando la ventana mostrada en 4.5. La flecha gruesa muestra la salida real de la red. Esta salida será comparada con la salida deseada (en amarillo).

Capítulo 5

Experimentos

Los experimentos realizados con la red neuronal y su desempeño con alarmas reales se discuten a detalle en las secciones 5.1, 5.2, 5.3 y 5.4. En la sección 5.5 se presenta una segunda serie de experimentos con datos sintéticos diseñados para probar el desempeño de la red neuronal con secuencias que contienen patrones reportadas en la literatura (periódicas y semiperiódicas con diferentes niveles de ruido). También se ha incluido en el apéndice B el código utilizado para todos los experimentos. **A las alarmas se les ha asignado un dígito de** 0 **a** 6**.**

La primer pregunta a responder es ¿qué tanta información de la secuencia pasada se requiere para predecir?

Tamaño de la ventana. El experimento 5.4 demuestra que para este conjunto de alarmas reales, el tamaño de la ventana puede variar sin perdida de desempeño y que por lo tanto, tenemos flexibilidad para escoger la cantidad de información de la secuencia para predecir la siguiente alarma. La figura 5.13 muestra la precisión alcanzada por la red neuronal con varios tamaños de ventana.

Selección del conjunto de entrenamiento. Los experimentos 5.1 y 5.2 muestran el efecto negativo que tiene la manera en que se escogen los ejemplos de entrenamiento en el desempeño de la red neuronal. La hipótesis inicial era que se debía escoger respetando la temporalidad, es decir, debíamos entrenar con el primer 80 % de la secuencia, y probar con el 20 % restante. La conclusión de los experimentos es que conviene usar muestro estratificado.

Una vez afinados estos importantes detalles, los experimentos mostrados en 5.3 corroboran que es posible anticiparse hasta en un 83 % a las fallas presentes en el historial de fallas reales que se usaron en esta tesis.

Por otro lado, los experimentos diseñados en la sección 5.5 corroboran que la red neuronal se desempeña tal y como se espera en otros escenarios concebibles y

reportados en la literatura. Por ejemplo, para una secuencia periódica, la red neuronal descubre el patrón y es capaz de predecir en el 100 % de los casos(véase 5.5.2). Si por el contrario se entrena a la red neuronal con una secuencia sin estructura aparente (por ejemplo los dígitos de π) la precisión de la red tiene un valor esperado: 1/10 (lo cual se muestra en 5.5.4). En estos experimentos se buscó evaluar la eficiencia de la red neuronal en predicción de eventos raros, predicción de eventos periódicos y semiperiódicos y predicción de ráfagas (*burst detection*).

Finalmente, **la comparación con otras técnicas**, presentada en la sección 5.6, corrobora que en la mayoría de los escenarios la red neuronal tienen mejor desempeño respecto a las técnicas comparadas.

La tabla 5.45 compara el desempeño de la red neuronal con: *Naive Bayes, Random Forest*, regresión logística y árboles de decisión.

Test & Score Tue

Settings

Sampling type: Stratified 10-fold Cross validation
Target class: Average over classes

Scores

Method	AUC	CA	F1	Precision	Recall
Neural Network	0.929	0.850	0.841	0.840	0.850
Tree	0.929	0.848	0.841	0.841	0.848
Logistic Regression	0.927	0.845	0.837	0.835	0.845
Random Forest	0.925	0.836	0.820	0.807	0.836
Naive Bayes	0.921	0.812	0.808	0.806	0.812

Figura 5.1: Comparación del desempeño con otros modelos. Hemos utilizado el criterio de F1 para ordenar el desempeño de los modelos de mayor a menor, si bien dicho orden coincide con el orden proporcionado por los criterios AUC y CA.

Obsérvese que en la comparación con otras técnicas de aprendizaje automatizado que se hizo en esta tesis (figura 5.45), la red neuronal tiene el mejor desempeño. Sin embargo, para ciertas clases, como la clase PANSL1F (etiquetada como clase cero en los experimentos), el modelo *TREE* o árbol de decisión se desempeña ligeramente mejor (ver figura 5.2). Cabe aclarar que no se hizo un análisis exhaustivo sobre los parámetros de los otros modelos ni un preprocesamiento especial de los datos.

Test & Score

Settings

Sampling type: Stratified 10-fold Cross validation
Target class: 0

Scores

Method	AUC	CA	F1	Precision	Recall
Tree	0.847	1.000	0.385	0.400	0.370
Logistic Regression	0.907	1.000	0.318	0.412	0.259
Naive Bayes	0.933	0.999	0.290	0.247	0.352
Neural Network	0.911	1.000	0.251	0.328	0.204
Random Forest	0.898	1.000	0.000	0.000	0.000

Figura 5.2: Tomando el criterio de F1 para comparar los modelos, para la clase PANSL1F (o clase etiquetada como 0) el modelo *TREE* se desempeña ligeramente mejor

5.1. Experimento 1

La figura 5.3 muestra las curvas de error y precisión[1]. La matriz de confusión se muestra en la figura 5.4.

La matriz de confusión mostrada en la imagen 5.4 nos indica que las alarmas 1 y 4 no fueron tomadas en cuenta como *target* (debido al muestreo que hemos considerado en la sección 4.6). Además notamos que algunas alarmas no fueron consideradas como respuesta de la red neuronal (no aparecen como nombre de un renglón). En la figura 5.5, se observa la desigualdad en la distribución de las alarmas *target* . Ante esta situación podríamos optar por seleccionar los datos de test (y por ende los de entrenamiento) de diferente manera.

Para los datos de test se obtiene un error de 0.4701603 y un porcentaje de alarmas predichas correctamente de 89.72846 % (porcentaje de precisión).

[1]La precisión es el cociente de las alarmas predichas correctamente entre el total de alarmas usadas para entrenamiento. Corresponde con la suma de la diagonal de la matriz de confusión entre el total de alarmas usadas como *target*.

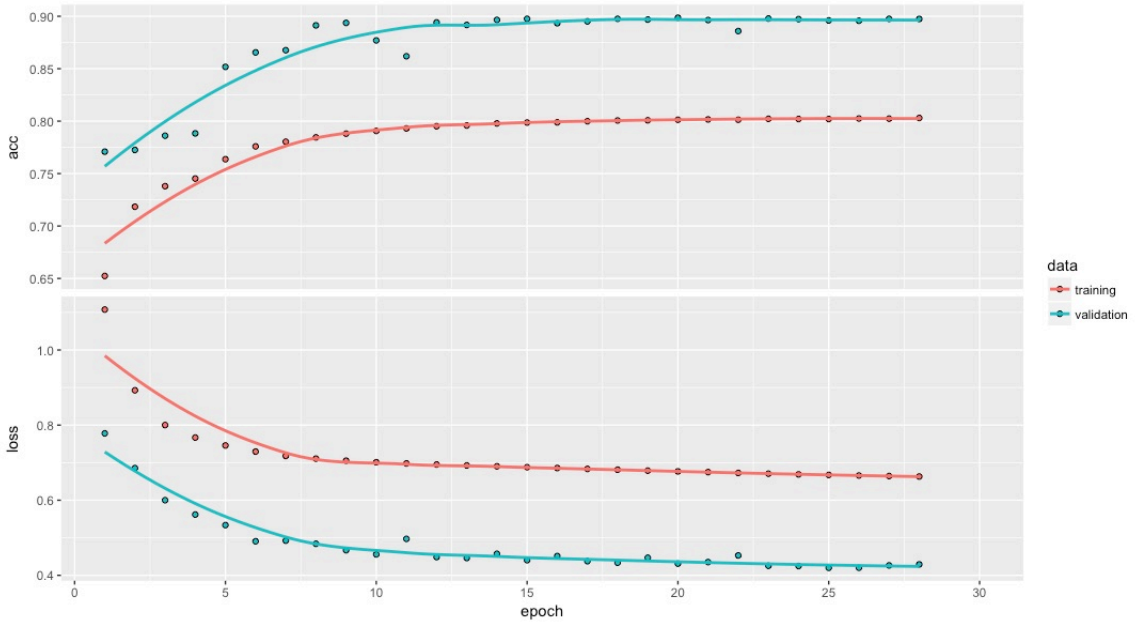


Figura 5.3: Curvas de error y precisión. No se ejecutaron las 30 épocas porque se activó el criterio de paro

ventanas.testtarget					
classes	0	2	3	5	6
1	31	24	28	10	30
2	6	43155	178	1152	1990
3	13	153	2530	59	162
4	0	0	1	3	1
5	1	188	28	2651	109
6	12	765	164	793	3213

Figura 5.4: Matriz de confusión producto del entrenamiento mostrado en 5.3 . **El renglón correspondiente a la alarma etiquetada como 0 se omite porque no es propuesta como salida por la red neuronal. Dicho renglón serían ceros.**

El siguiente experimento explora una selección distinta.

5.2. Experimento 2

Con el fin de tener mayor representatividad de las alarmas tomamos aleatoriamente el 20 % de la totalidad de los datos como *test*, en lugar de usar la última parte de la secuencia. Esto aumenta la posibilidad de representación de las alarmas que se presentan con menos frecuencia. El resultado de entrenar bajo este muestreo de ventanas se muestra en la figura 5.6. La matriz de confusión se muestra en la figura 5.7.

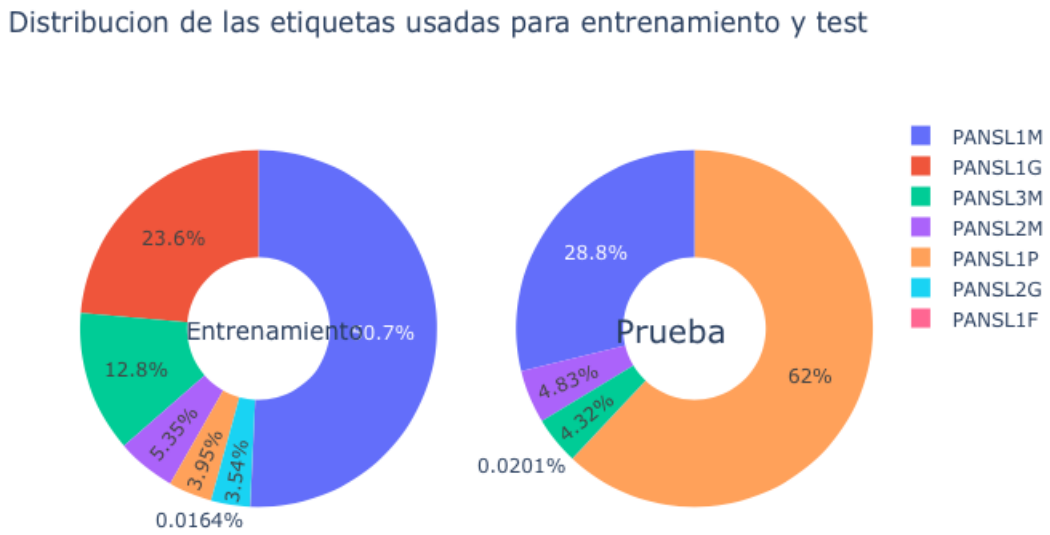


Figura 5.5: Porcentaje de las alarmas deseadas como salida de las ventanas del conjunto de entrenamiento y prueba o *test*. Algunas alarmas no están representadas en el conjunto de prueba.

Podemos ver en la matriz de confusión mostrada en 5.7 que ahora todas las alarmas son tomadas en cuenta en los datos de test. Esto también queda plasmado en la gráfica 5.9. Sin embargo, el desempeño global fue menor al obtenido en el experimento anterior.

La alarma que denotamos por 1 (etiquetada en la matriz de confusión como 0) sigue sin poder ser predicha como se muestra en la imagen 5.8.

Para evaluar el desempeño de la red, los ejemplos de *test* contemplaron todos los *targets* posibles en la proporción mostrada en la imagen 5.9.

En este caso **para los datos de test se obtuvo un error de 0.6129664 y un porcentaje de precisión de 80.49 %.**

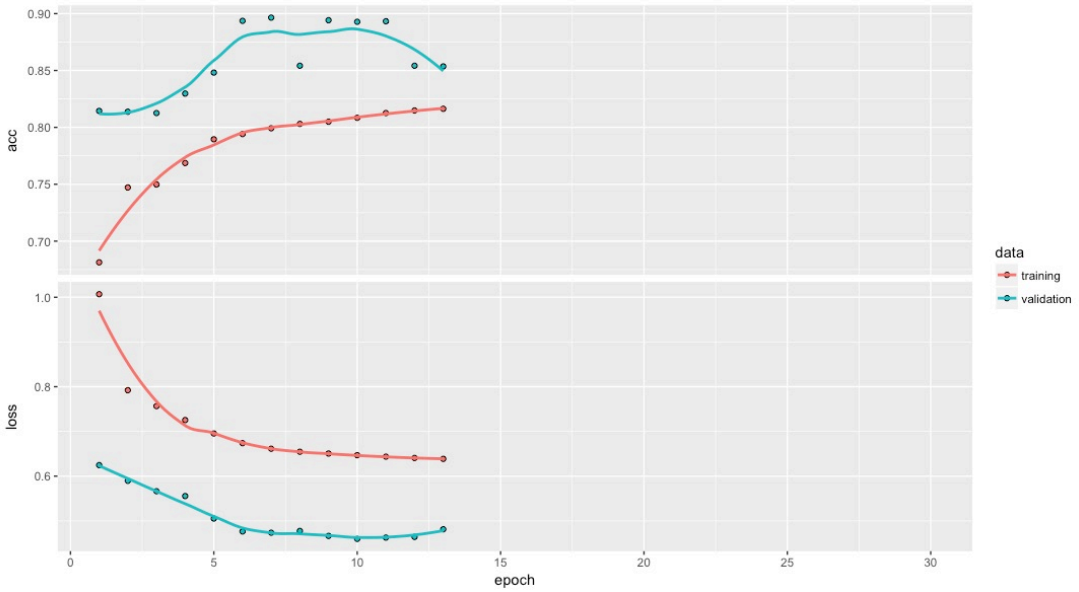


Figura 5.6: Curvas de error y precisión. No se ejecutaron las 30 épocas porque se activó el criterio de paro

ventanas.testtarget							
classes	0	1	2	3	4	5	6
1	8	8323	535	65	204	125	1301
2	1	535	31808	165	506	1028	1578
3	3	24	168	2607	7	18	92
4	0	6	1	3	2	3	10
5	0	19	37	0	320	2	36
6	5	748	988	154	412	2093	3475

Figura 5.7: Matriz de dispersión. **El renglón correspondiente a la alarma etiquetada como** 0 **se omite porque no es propuesta como salida por la red neuronal. Dicho renglón serían ceros.**

alarma1	alarma2	alarma3	alarma4	alarma5	alarma6	alarma7
0.0000	0.8620	0.9484	0.8707	0.0014	0.0006	0.5353

Figura 5.8: Porcentaje de alarmas predichas correctamente por la red.

5.3. Experimento 3

Exploramos si eliminar el criterio de paro puede mejorar el desempeño de la red. Seleccionamos al azar el 80 % de las ventanas para el conjunto de entrenamiento y aumentamos a 50 épocas. Los resultados se muestran en la imagen 5.10.

La matriz de confusión que se obtiene con los ejemplos de *test* se muestra en la imagen 5.11.

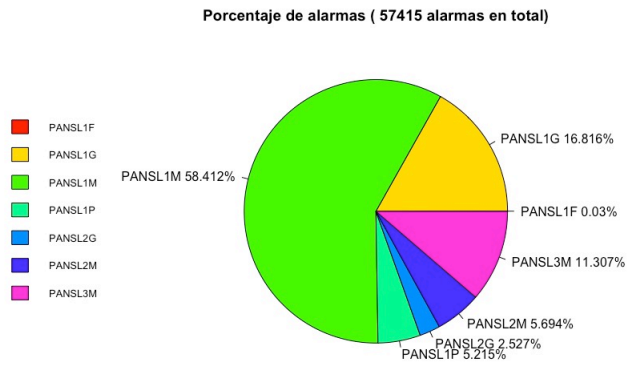


Figura 5.9: Porcentaje de alarmas que fueron utilizadas como salidas deseadas en los datos de test.

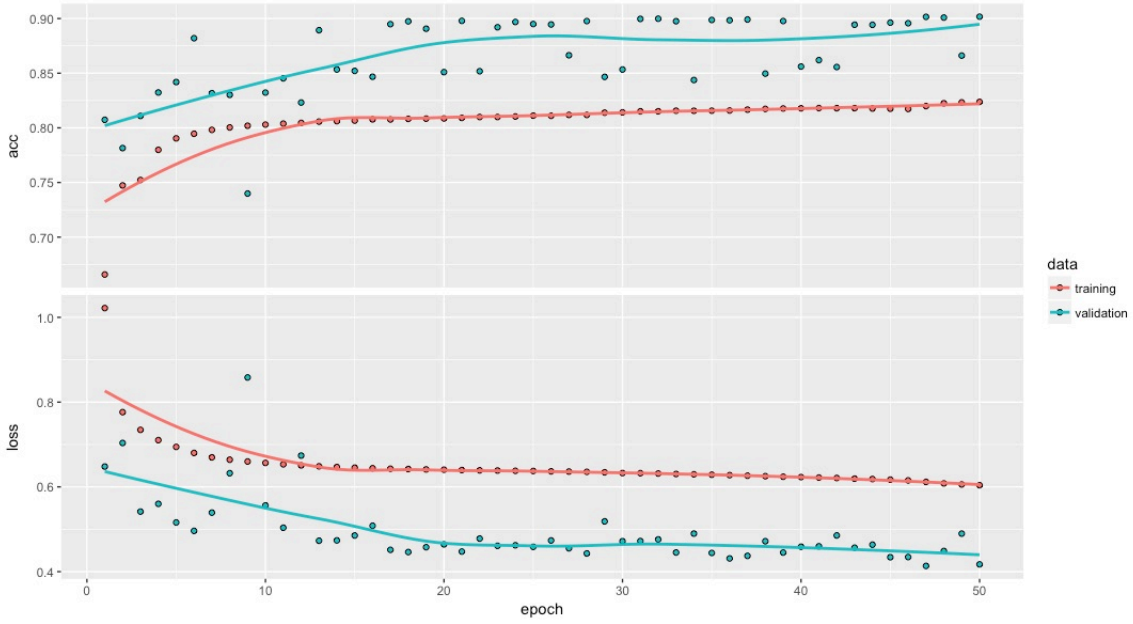


Figura 5.10: Entrenamiento a lo largo de 50 épocas con una ventana de tamaño 5. Se eliminó el criterio de paro.

Además notamos que los porcentajes de precisión de cada alarma son mejores que cuando incluimos el criterio de paro de 3 épocas sin mejora sustancial de 0.001 en la función error o *cost function*. Esto se muestra en la imagen 5.12.

	ventanas.testtarget						
classes	0	1	2	3	4	5	6
1	8	7914	476	44	182	98	992
2	1	552	31769	185	473	625	1793
3	3	33	180	2620	36	24	120
4	0	61	66	40	125	17	192
5	1	108	530	33	338	2334	265
6	1	874	483	70	291	143	3270

Figura 5.11: Matriz de dispersión. Se completaron 50 épocas, se utilizó una ventana de tamaño 5 y se tomaron al azar las ventanas empleadas para probar el modelo. **El renglón correspondiente a la alarma etiquetada como** 0 **se omite porque no es propuesta como salida por la red neuronal. Dicho renglón serían ceros.**

alarma1	alarma2	alarma3	alarma4	alarma5	alarma6	alarma7
0.0000	0.8294	0.9482	0.8757	0.0865	0.7201	0.4931

Figura 5.12: Porcentaje de precisión alcanzado para cada alarma en los ejemplos de prueba o *test*.

Se obtiene un error de 0.4172 y una precisión de 90.7 % en los datos de validación al termino de las 50 épocas; en los datos de test un error de 0.5655353 y una precisión de 83.72 %.

5.4. Experimento 4

Graficamos la precisión alcanzada para cada alarma y la precisión promedio para diferentes tamaños de ventana. Queremos observar el cambio en la precisión de las predicciones al modificar el tamaño de ventana.

Para cada tamaño de ventana n (desde 5 hasta 100), tomamos el 80 % del total de ventanas al azar para entrenar la red y realizamos 15 épocas de entrenamiento. Para los ejemplos de test, se grafica el promedio de las precisiones obtenidas por cada alarma. Continuamos hasta llegar a los resultados para la ventana de tamaño 100. Los resultados se muestran en la imagen 5.13. **Se observa un decaimiento en la precisión a medida que el tamaño de ventana es mayor.**

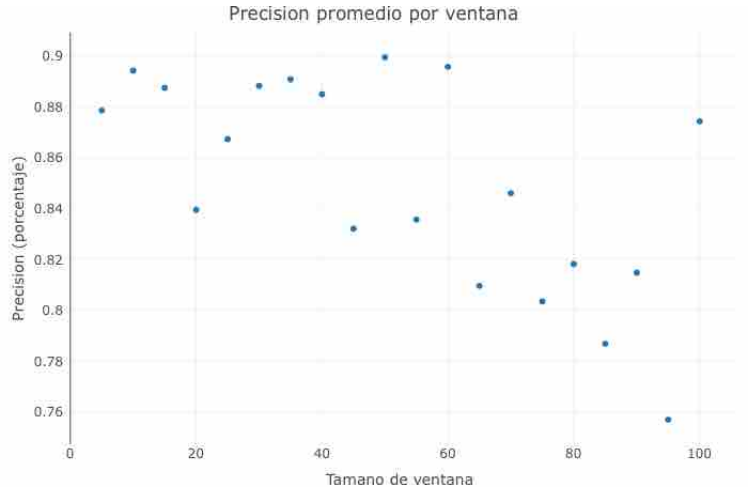


Figura 5.13: Porcentaje de precisión general por cada tamaño de ventana. **Se observa un decaimiento en la precisión a medida que el tamaño de ventana es mayor.**

5.5. Experimentos de control

La red neuronal presentada en el apéndice anterior logró una precisión de clasificación mayor al 80 % en el conjunto de ejemplos de test. Sin embargo, podemos preguntarnos si una precisión similar puede alcanzarse con otras secuencias. Para responder esta pregunta se diseñaron algunas secuencias artificiales con las que se pone a prueba una red con la misma arquitectura. **Se aclara que se ha tomado la libertad de crear secuencias con más dígitos ya que el interés está puesto en la capacidad de clasificación de la red neuronal en secuencias sintéticas**.

5.5.1. Experimento de control: función cuadrática

Se presenta una secuencia de números generada por la función x^2. El i-ésimo término de la secuencia viene dado por la función:

$$f(i) = i^2 \mod 7 \tag{5.1}$$

Se realiza un experimento con 15 épocas y con una ventana de tamaño 5. La evolución del entrenamiento se muestra en la imagen 5.14. Notamos que el error cae rápidamente a cero y la precisión sube a uno.

De los resultados obtenidos podemos concluir que la estructura de la secuencia no plantea un reto para la red neuronal. La secuencia no posee en su estructura azar y el número de elementos que la conforman es muy pequeño; de hecho, la secuen-

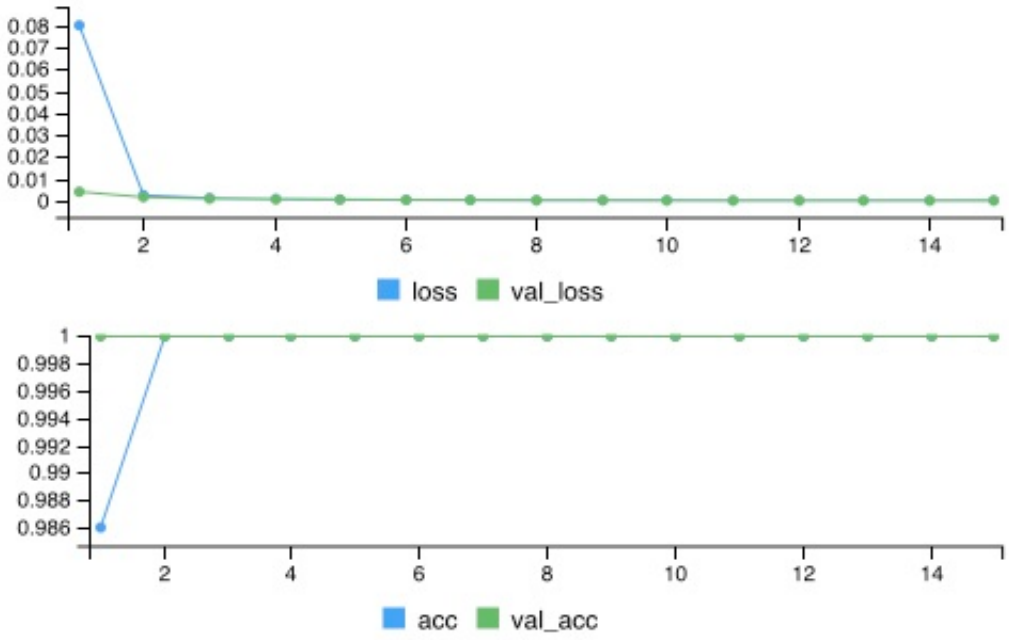


Figura 5.14: Entrenamiento de la red usando la secuencia descrita anteriormente. Sobre el conjunto de *test* o prueba se obtiene un error de 0.0001509306 y una precisión de 1.

cia está conformada por repeticiones de los dígitos 0,1,2 y 4, algo que podía observarse por la definición de al función que genera la serie.

5.5.2. Experimento de control: secuencia 1-20

Podemos introducir una secuencia de números que coincida con los números naturales hasta cierto número, digamos 19. La lista de alarmas consiste en repetir el patrón $1, 2, 3, ..., 19, 0, 1, 2, ..., 19, 0, 1, 2..., 19, 0, 1....$ La longitud de la secuencia que utilizaremos para hacer un experimento de control que contemple este patrón es de $10,256$.

Sin seleccionar al azar los ejemplos que serán utilizados como ejemplos de test y los de entrenamiento (de los cuales el último 20 % se utilizará como datos de validación) y además seleccionando un tamaño de ventana de 5, se procede a graficar el error y la precisión obteniéndose la grafica que se muestra en la imagen 5.15.

Modificamos el número de épocas para este experimento a 500 y la historia del entrenamiento se muestra en la imagen 5.16.

Modificando el tamaño de ventana a 20, haciendo 15 épocas y con una lista de alarmas de 10,000 se obtiene la gráfica 5.17 que muestra el proceso de entrenamiento.

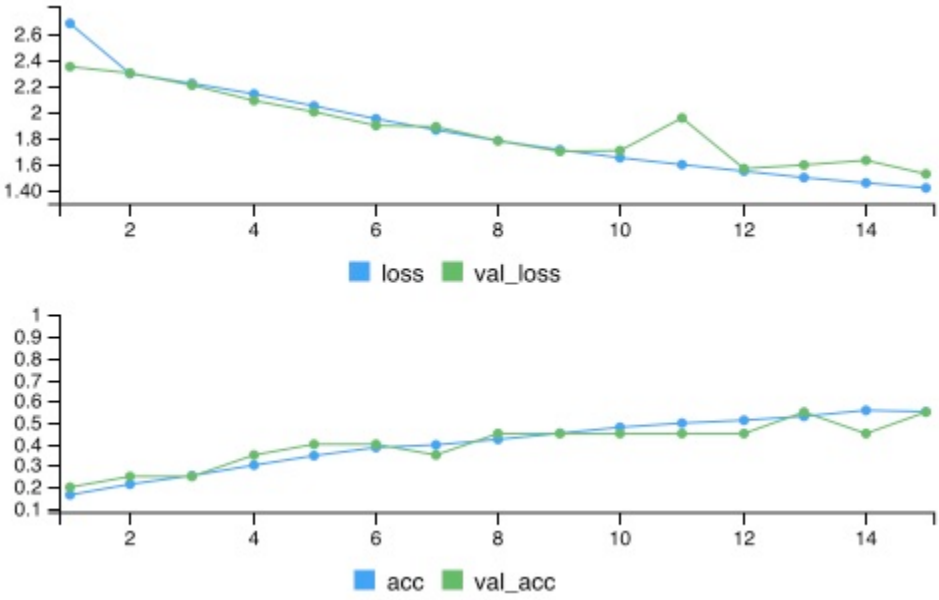


Figura 5.15: Gráfica del error a lo largo de las 15 épocas. Se observa una tendencia clara que nos indica que es posible continuar el entrenamiento para que la función costo sea menor. Sobre los ejemplos de prueba se obtiene una precisión de clasificación de 0.5497749 y un error de 1.520408.

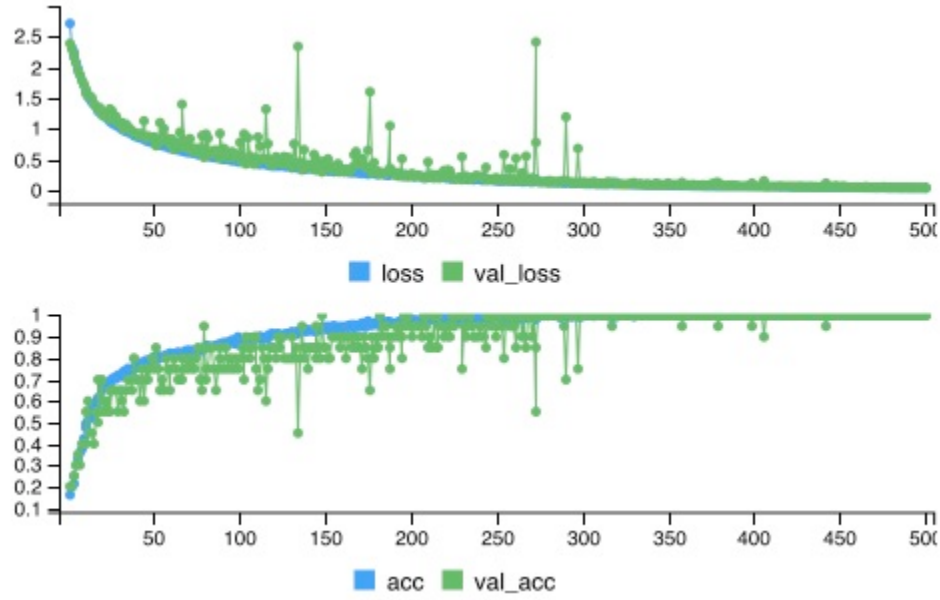


Figura 5.16: Gráfica del error a lo largo de las 500 épocas. Se alcanza el 100 % en la precisión de los datos de validación. Sobre los ejemplos de prueba se tiene un error de 0.04505581 y una precisión de 100 %

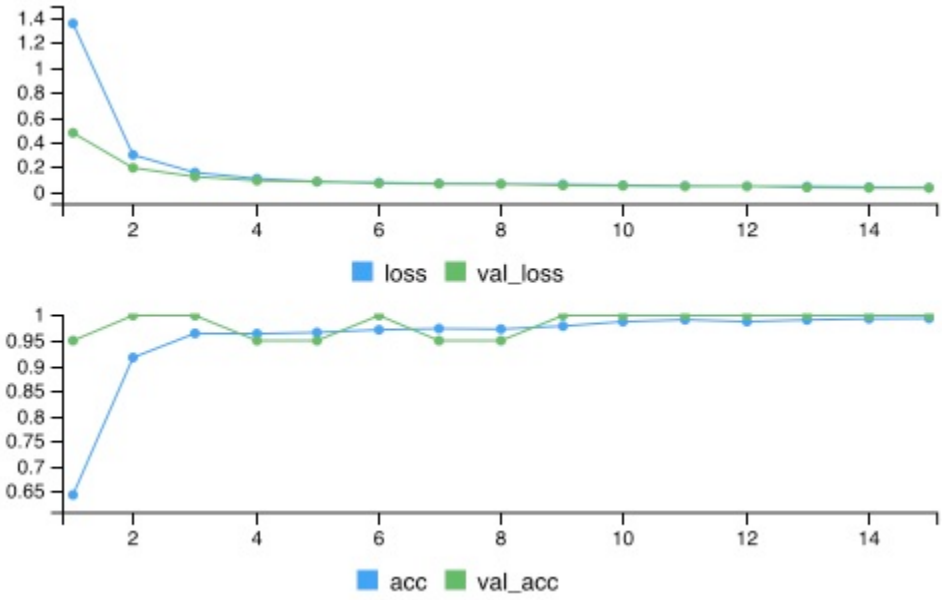


Figura 5.17: Grafica del error y precisión a lo largo de las 15 épocas y con un tamaño de ventana de 20. Se alcanza el 100 % en la precisión de los datos de validación. Sobre los ejemplos de prueba se tiene un error de 0.03107293 y una precisión de 100 %.

Si bien los resultados obtenidos en 5.17 son excelentes, la red tuvo desaciertos en algunas etiquetas como lo muestra la matriz de confusión que se muestra en 5.18.

ventanas.testtarget																				
classes	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100

Figura 5.18: Matriz de confusión obtenida final del entrenamiento mostrado en la imagen 5.17.

Si fijamos el número de épocas a 15 y usamos un tamaño de ventana de 5

pero modificamos el número de alarmas a 287, 256 obtenemos los resultados mostrados en 5.17. Notamos que hubo algunos errores en las etiquetas, lo cual se refleja en la matriz de confusión 5.18

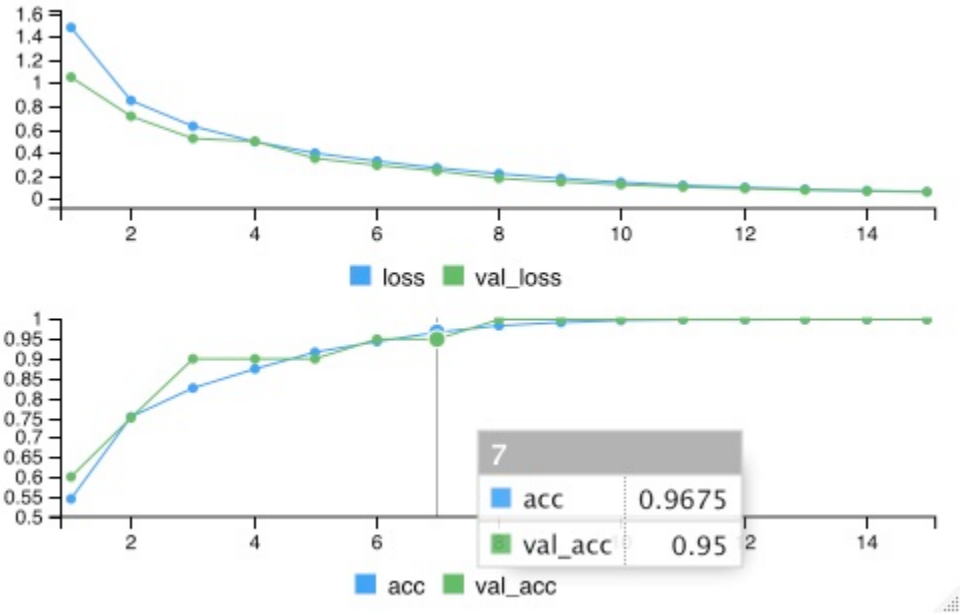


Figura 5.19: Grafica del error a lo largo de las 30 épocas, con una lista de alarmas de 287,256 y con un tamaño de ventana de 5. Se alcanza el 100 % en la precisión para los ejemplos de validación. Para los datos de prueba, obtenemos un error de 0.06319334 y una precisión de 100 %.

ventanas.testtarget

classes	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	2872	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	2872	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	2872	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	2872	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	2872	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	2872	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	2873	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	2873	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	2873	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	2873	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	2873	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	2873	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	2873	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	2873	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2873	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2873
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 5.20: Un extracto de la Matriz de dispersión que se obtiene en el entrenamiento llevado a cabo en 5.19.

Conclusión: A pesar de ser una secuencia que cualquier humano podría predecir, podríamos concluir que la cantidad de ejemplos de la que dispongamos influye en la rapidez con la que clasificamos nuestros ejemplos (véase las gráficas correspondientes a los entrenamientos) . Además, la longitud del ciclo influye en la rapidez con la que alcanzamos la precisión máxima en los ejemplos de test.

5.5.3. Experimento de control: dígitos aleatorios del 1 al 7 producidos por una distribución uniforme

Las secuencias anteriores son bastante sencillas pues tienen un comportamiento periódico. En este experimento buscamos poner a prueba la red neuronal con una secuencia más azarosa. Para probar la capacidad de predicción de nuestro enfoque basado en ventanas con la red neuronal, creamos una lista de dígitos aleatorios (de cierta longitud) y los guardamos en un vector. En nuestro ejemplo, dichos dígitos son del 1 al 7. Una lista así tiene un aspecto como el que sigue: $lista = (1, 7, 4, 5, 3, 7, 7, 1, ...)$ donde cada uno de los dígitos de la lista es tomado aleatoriamente del conjunto de $A = \{1, 2, 3, ..., 7\}$ de manera uniforme (cada dígito tiene la misma posibilidad de ser seleccionado).

Con una ventana de tamaño 5 obtenemos la grafica que se muestra en la imagen 5.21.

Conclusión: Notamos que la precisión alcanzada es un resultado esperado, $\frac{1}{7} \approx 0.14$. Esto porque al ser cada dígito generado de manera aleatoria por una distribución uniforme, cada uno tiene una probabilidad de aparecer de $\frac{1}{7}$. Tal parece que la red neuronal intenta adivinar el dígito como lo haría una persona escogiendo un número aleatorio. Ambos, después de muchos intentos, tenderían a una precisión de $\frac{1}{7}$.

5.5.4. Experimento de control: Dígitos del número pi

Podemos experimentar con algunos secuencias que impliquen un reto mayor a la red neuronal: la secuencia de los dígitos del número π (pi). Se elabora una lista de los primeros 10,000 dígitos de π y a partir de esa lista se hacen predicciones usando la red neuronal. Los dígitos se distribuyen como se muestra en la imagen 5.22.

Los resultados obtenidos durante el entrenamiento de la red neuronal se muestran en 5.23.

En 5.24 se grafica la precisión obtenida sobre los ejemplos de prueba o *test* cambiando la longitud de la ventana.

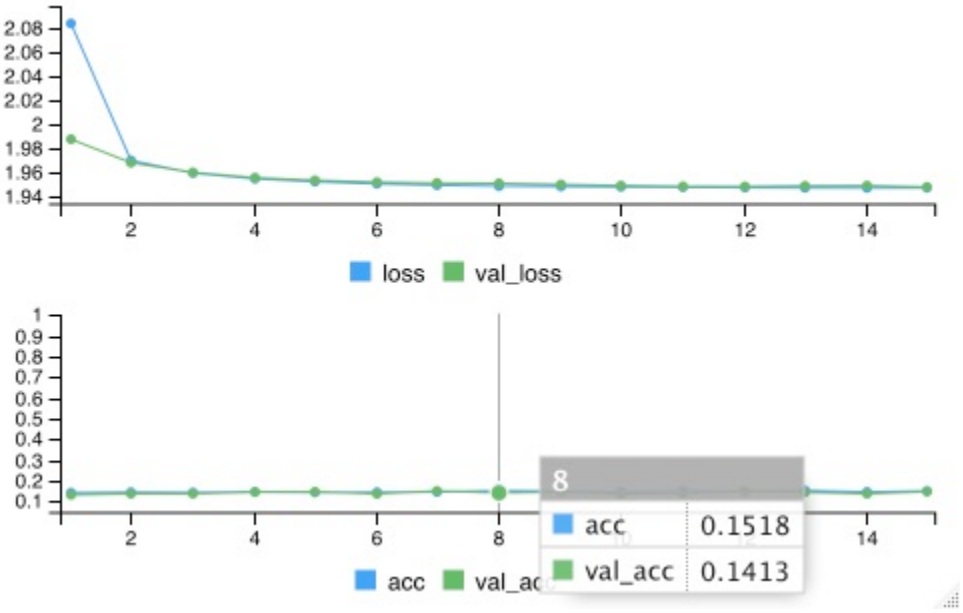


Figura 5.21: Grafica del error a lo largo de las 15 épocas, una lista de dígitos de $10,000$ y con un tamaño de ventana de 5. No hay mejoras significativas con el transcurrir de las épocas. Sobre los ejemplos de prueba se obtiene un error de 1.948352 y una precisión de 0.1465733, cifras no muy distintas para el conjunto de entrenamiento en la época 15.

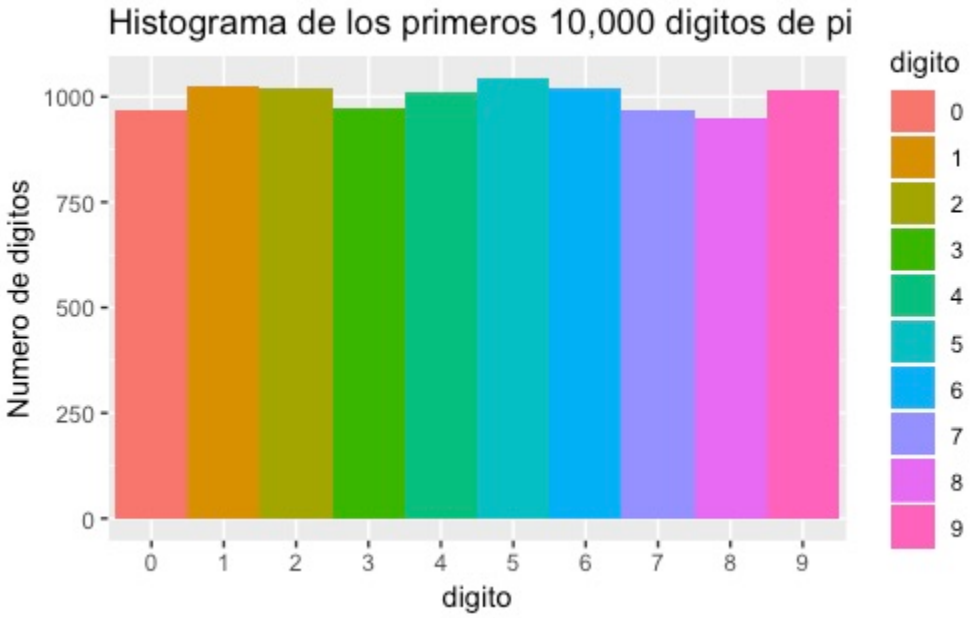


Figura 5.22: Histograma de los primeros 10,000 dígitos de π.

Podemos extender el número de ventanas y graficar el porcentaje de precisión por alarma. Esto se muestra en 5.25.

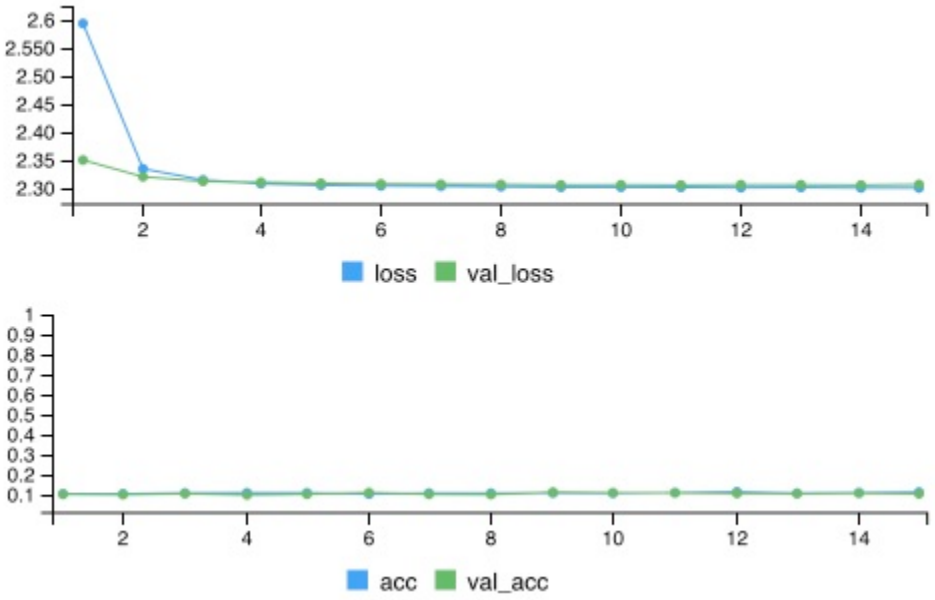


Figura 5.23: Grafica del error a lo largo de 15 épocas, una lista de dígitos de $10,000$ y con un tamaño de ventana de 5. En los ejemplos de prueba se obtiene una precisión de 0.09504752 y un error de 2.310056.

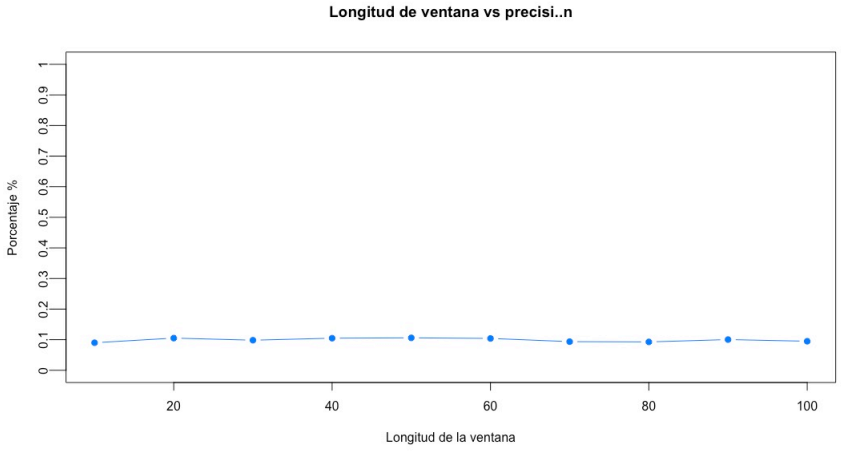


Figura 5.24: Gráfica del porcentaje de aciertos (precisión) obtenidos por la red neuronal al cambiar el tamaño de las ventanas en los ejemplos de prueba. En general se logra una presición de 10 % aproximadamente. En cada ventana se entrenó la red con 30 épocas.

5.5.5. Experimento de control: Secuencia periódica perturbada

En el siguiente experimento se introduce una secuencia identificable pero con un dígito que aparece de manera azarosa. Buscamos que la frecuencia de aparición del

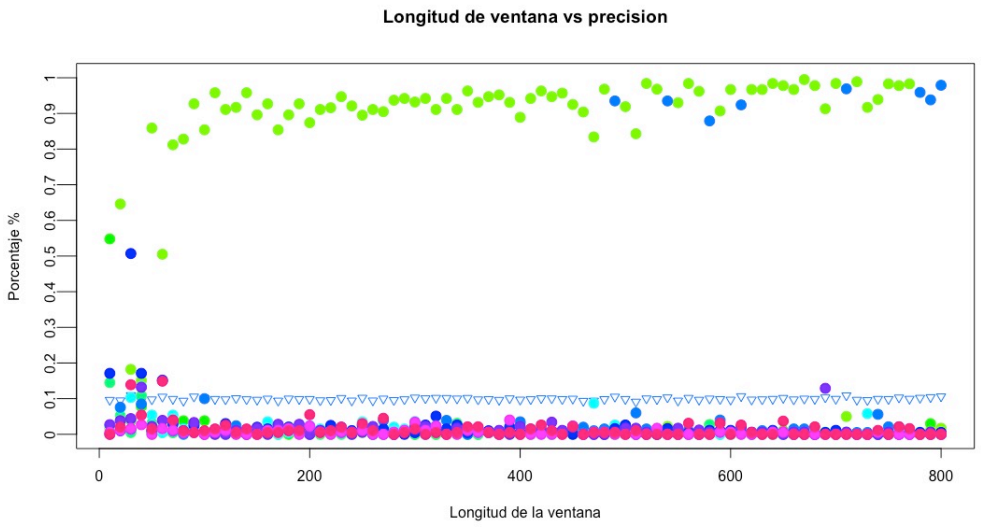


Figura 5.25: Gráfica del porcentaje de aciertos obtenidos por la red neuronal (precisión) en los ejemplos de prueba al cambiar el tamaño de ventana. El triangulo azul indica la precisión promedio y se muestra casi constante (10 % aproximadamente). En cada ventana se entrenó a la red con 30 épocas a lo mucho, pues hubo criterio de paro si percibió sobreajuste (paciencia 3 y 0.00001 como valor delta). La red tuvo mucho éxito en predecir un dígito que es el cero (en verde).

dígito sea muy pequeña. Una secuencia que tenga un periodo (pensemos de 7) del estilo x = (0,1,2,...,5,6,0,1,2,...,5,6,0,1,...) pero en la que, de manera azarosa, introducimos un dígito que no tenga relación con la secuencia, digamos un 7. Si bien la secuencia sigue una estructura predecible, hay entradas en la lista que en vez de presentar el dígito esperado, nos presenta un 7. Por ejemplo $x = (0, 1, 2, 3, 7, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, ..)$. Establecemos que el dígito 7 tenga una probabilidad de aparición en un índice de la lista de $\frac{108}{287256} \approx 0.00037$. La distribución de los 10,000 dígitos se muestra en la imagen 5.26.

Establecemos el tamaño de ventana en 5 y realizamos 15 épocas de entrenamiento. Se obtienen los resultados mostrados en la imagen 5.27.

A pesar de los resultados obtenidos, buenos en general, el dígito 7 no fue propuesto por la red como salida de una ventana. Lo cual se puede ver en la matriz de dispersión que se muestra en la imagen 5.28.

Las etiquetas de test presentaban la distribución mostrada 5.29.

La gráfica del entrenamiento de la red sugiere que no necesitamos muchas épocas para lograr una buena precisión en general (más no precisión en el dígito azaroso). Además, en la matriz de dispersión también se observa que si bien clasificó muchos dígitos correctamente, la red cometió algunos pequeños errores en los dígitos de alguna manera predecibles.

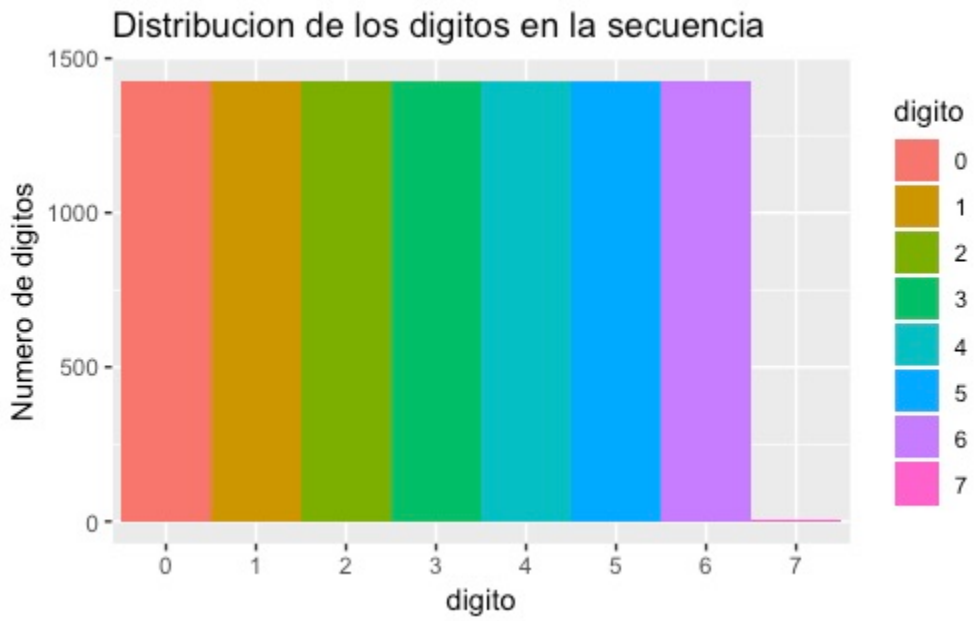


Figura 5.26: Histograma de los dígitos en la secuencia. El dígito 7 aparece en raras ocasiones.

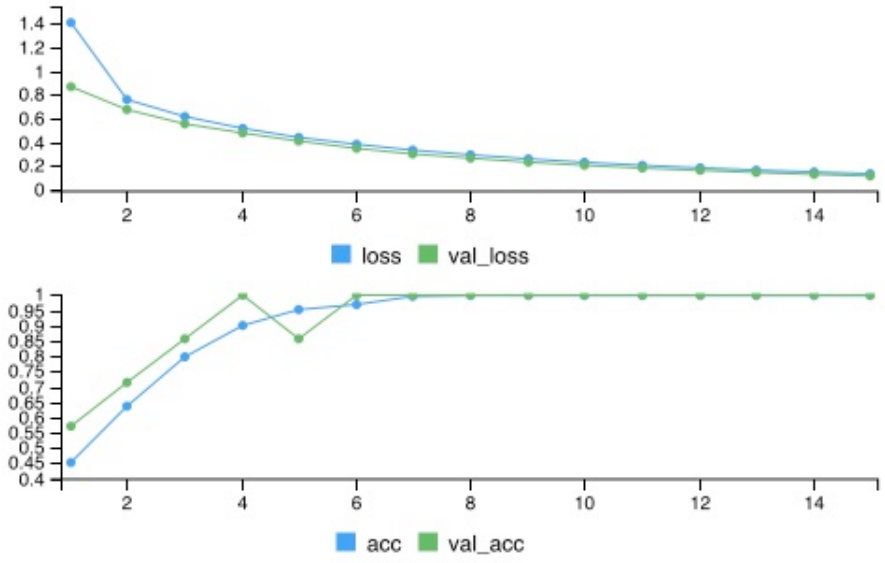


Figura 5.27: Entrenamiento de la red a lo largo de 15 épocas. Se obtiene una precisión del 100 % en los datos de validación al finalizar la época 15. Sobre el conjunto de prueba se obtiene una precisión de 0.997999 y un error de 0.1365842.

Se grafica ahora la precisión sobre los ejemplos de prueba como función del tamaño de ventana y se obtiene los resultados mostrados en 5.30.

ventanas.testtarget								
classes	0	1	2	3	4	5	6	7
0	286	0	1	0	0	0	0	0
1	0	285	0	0	0	0	0	1
2	0	0	285	0	0	0	0	0
3	0	0	0	286	0	1	0	0
4	0	0	0	0	284	0	0	0
5	0	0	0	0	0	284	0	0
6	0	0	0	0	1	0	285	0

Figura 5.28: Matriz de dispersión. El dígito 7 no fue propuesto por la red como *output* de alguna ventana en los ejemplos de prueba, por lo que no aparece como renglón.

digito	cantidad
<chr>	<int>
0	1427
1	1429
2	1428
3	1429
4	1428
5	1427
6	1428
7	4

Figura 5.29: Distribución de las 30, 015 etiquetas de los ejemplos usados para prueba.

5.5.6. Experimento de control: Secuencia periódica con mayor ruido

Presentamos ahora una secuencia de ocho dígitos en la cual un dígito es introducido a manera de ruido en la secuencia. Es muy similar a la secuencia presentada anteriormente con la salvedad de que ahora el dígito introducido como ruido tendrá una mayor probabilidad de aparecer, $\frac{168587}{287256} \approx 0.58\,\%$ para ser precisos. La idea es observar qué tanto puede afectar a la predicción un dígito que aparezca de manera azarosa con esa probabilidad. La distribución de los dígitos de la secuencia se muestra en la imagen 5.31.

El historial de entrenamiento de la red neuronal, estableciendo un tamaño una ventana de longitud 5, se muestra en la imagen 5.32.

La distribución de las etiquetas de los ejemplos de *test* se muestran en 5.33.

La matriz de dispersión se muestra en la imagen 5.34.

Se utilizan más épocas en el entrenamiento. Elevamos el número de épocas a 500 y se obtiene el historial de entrenamiento mostrado en 5.35.

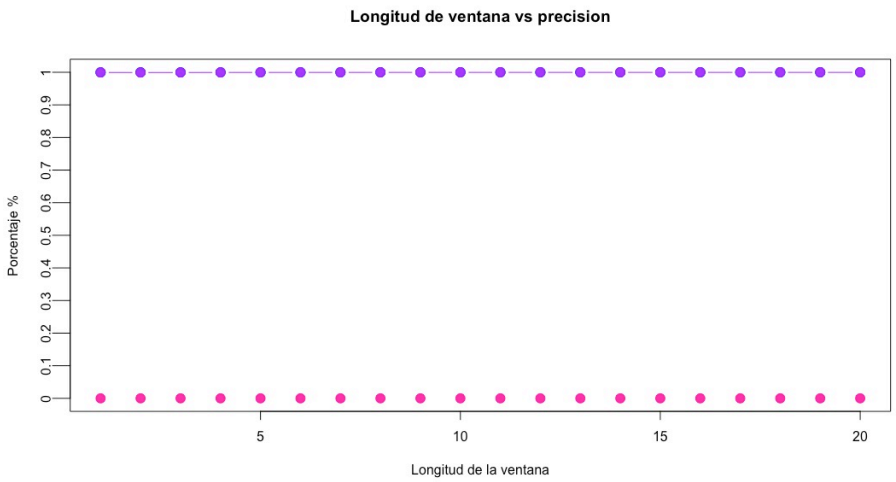


Figura 5.30: Historial de los porcentajes de precisión con diversos tamaño de ventana. Se utilizan 15 épocas como máximo y se para el entrenamiento bajo ciertas condiciones. Con todas se alcanza un 100 % de precisión, incluido el porcentaje de precisión general sobre los ejemplos de prueba a excepción del porcentaje de precisión correspondiente al dígito que fue introducido de manera azarosa, es decir 7 (cuya precisión fue de 0 %).

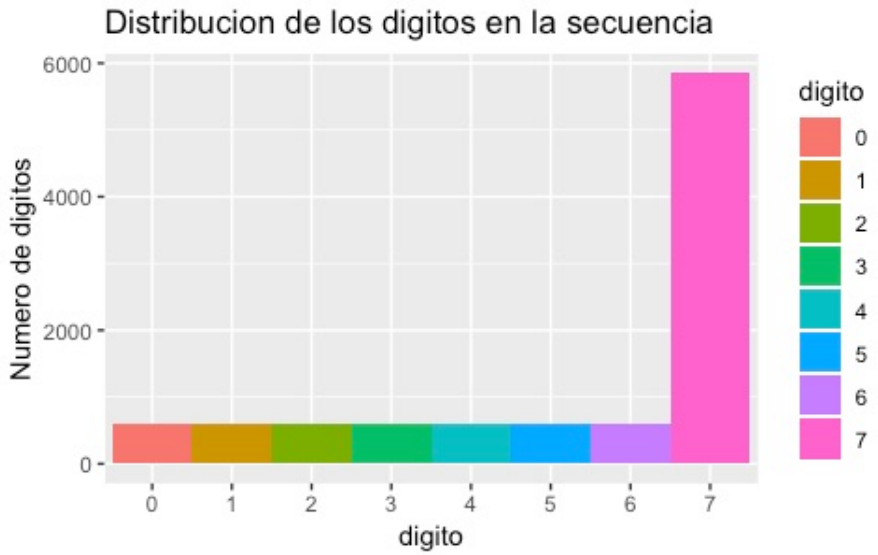


Figura 5.31: Distribución de los dígitos en la secuencia.

Se introduce el dígito 7 a la secuencia de una manera equitativa y se procede a entrenar la red neuronal con 15 épocas. La aparición de los dígitos en la secuencia se distribuye como se muestra en la imagen 5.36.

El entrenamiento de la red neuronal utilizando un tamaño de ventana de 5 se muestra en la imagen 5.37.

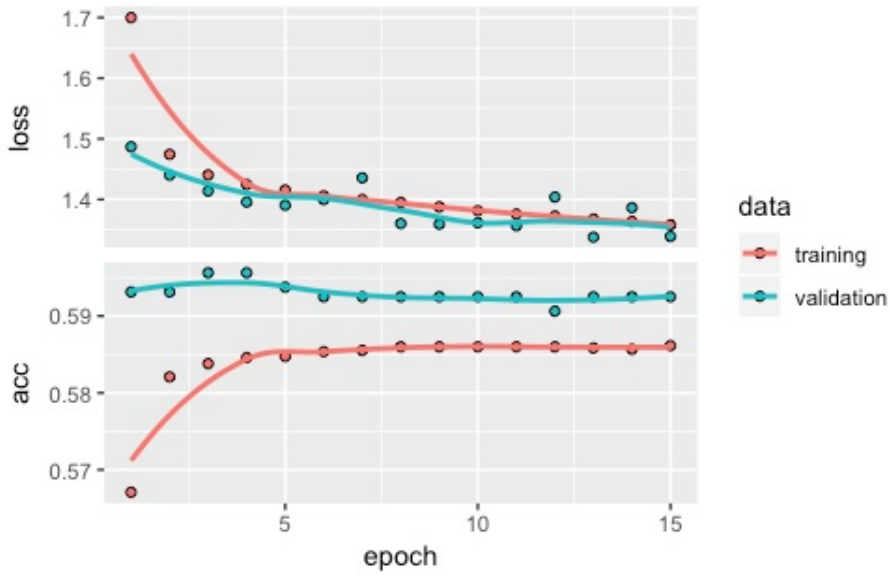


Figura 5.32: Se introduce en una secuencia el dígito 7 de manera aleatoria con una probabilidad relativamente alta de aparecer. En los ejemplos de prueba se obtiene un error de 1.380537 y una precisión de 0.5802901. En la gráfica se muestra que la precisión alcanzada en los ejemplos de validación es aproximadamente de 0.6 .

0	1	2	3	4	5	6	7
465	469	467	485	469	477	470	4694

Figura 5.33: Distribución de las etiquetas de los ejemplos de test.

	ventanas.testtarget							
classes	0	1	2	3	4	5	6	7
3	0	0	0	12	0	0	0	12
7	119	126	119	99	125	119	120	1148

Figura 5.34: Matriz de dispersión. **Las etiquetas de varias alarmas son omitidas como renglones porque no son propuestas por la red neuronal.**

La matriz de dispersión se muestra en la imagen 5.38.

5.6. Comparación con otros clasificadores

Los experimentos descritos hasta hora muestra un buen desempeño de la red neuronal, pero cabe la pregunta si otros clasificadores pueden tenerlo también. Los experimentos siguientes comparan el desempeño de la red neuronal con: *Naive Bayes, Random Forest*, regresión logística y árboles de decisión.

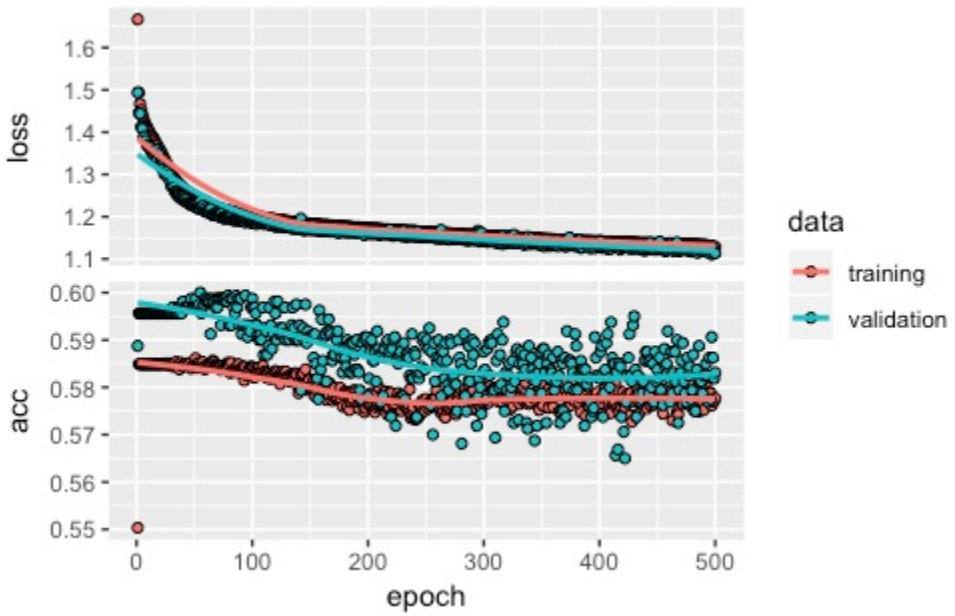


Figura 5.35: Entrenamiento a lo largo de 500 épocas. El error sobre el conjunto de prueba es de 1.15104 y su precisión es de 0.5587794. Observamos presencia de *overfitting*.

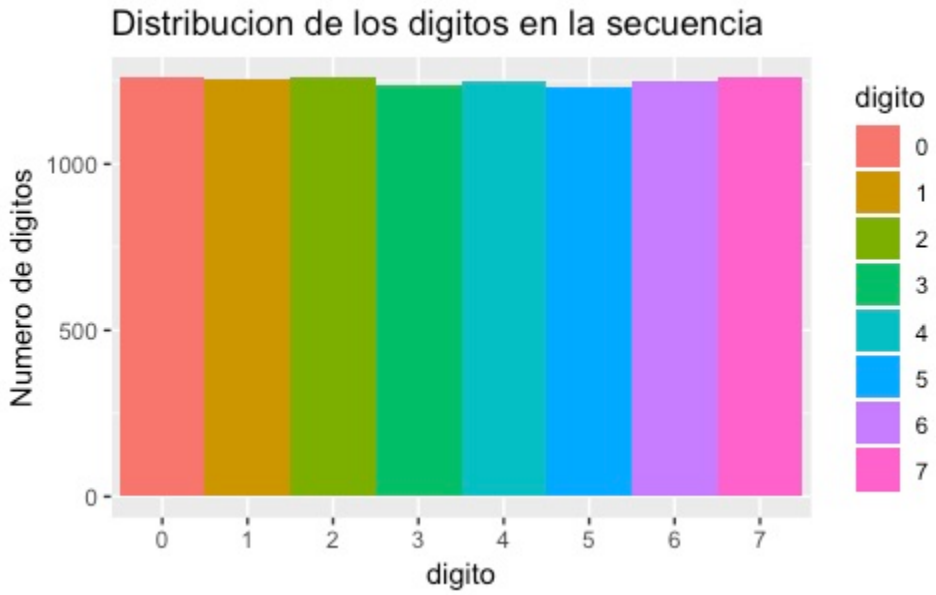


Figura 5.36: Distribución de los dígitos en la secuencia. El dígito introducido de manera azarosa aparece aproximadamente las mismas veces que los demás dígitos.

Por razones de simplicidad usaremos el software *Orange*[2]. Cabe aclarar que no se hizo un análisis exhaustivo sobre los parámetros de los otros modelos ni un

[2] `https://orange.biolab.si`. Una breve descripción del software se incluye en el apéndice A.

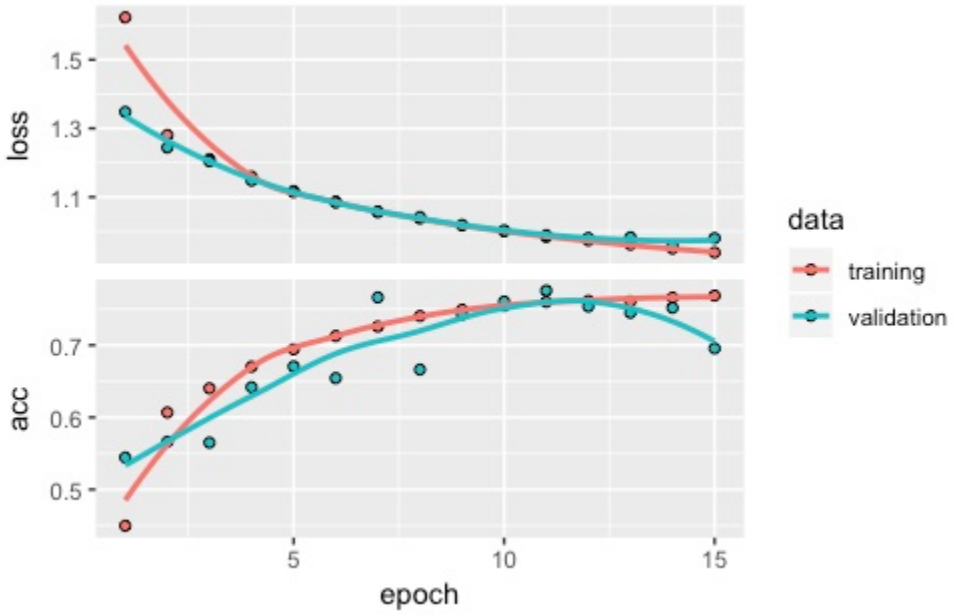


Figura 5.37: Entrenamiento de la red neuronal a lo largo de 15 épocas utilizando un tamaño de ventana de 5. Notamos la presencia de sobrejuste o *overfitting*. Sobre los ejemplos de prueba se obtiene un error de 0.9743934 y una precisión de 0.6853427.

ventanas.testtarget

classes	0	1	2	3	4	5	6	7
0	1.00	0.12	0.02	0.00	0.00	0.01	0.64	0.23
1	0.00	0.88	0.12	0.02	0.00	0.00	0.00	0.12
2	0.00	0.00	0.87	0.11	0.02	0.02	0.00	0.14
3	0.00	0.00	0.00	0.85	0.11	0.07	0.00	0.15
4	0.00	0.00	0.00	0.00	0.84	0.00	0.00	0.12
5	0.00	0.00	0.00	0.02	0.00	0.81	0.00	0.13
6	0.00	0.00	0.00	0.00	0.00	0.00	0.19	0.05
7	0.00	0.00	0.00	0.00	0.03	0.09	0.17	0.05

Figura 5.38: Matriz de dispersión al final del entrenamiento mostrado en 5.37. El porcentaje es con respecto del total de cada dígito presente en las etiquetas de prueba. *La suma de las columnas debe ser igual a* 1 *pero surgen errores por redondeo*

.

tratamiento especial a la hora de introducir los datos, por lo que es posible que existan mejoras en los resultados obtenidos en este trabajo.

Utilizaremos una regresión lógistica con los parámetros mostrados en 5.39.

Otra técnica a la cual se recurre comúnmente en tareas de clasificación es el árbol de decisión. Emplearemos uno con los parámetros especificados en la imagen 5.40.

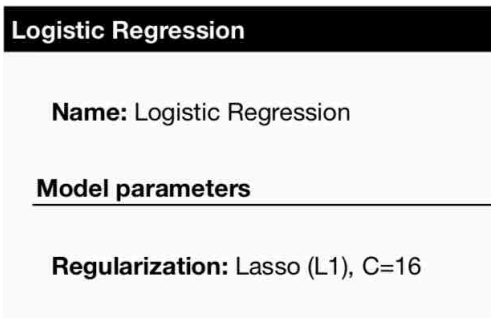


Figura 5.39: Parámetros del modelo regresión logística en Orange

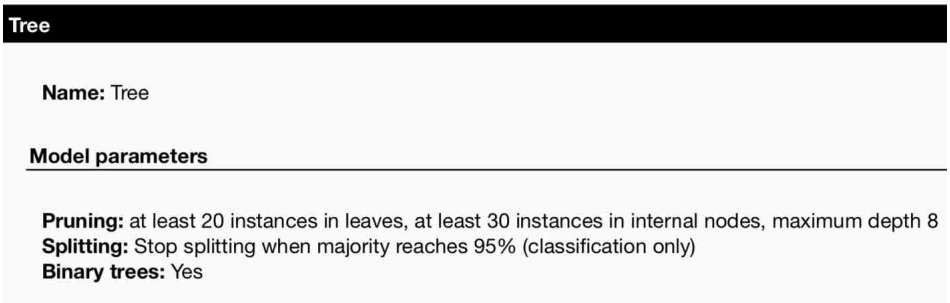


Figura 5.40: Parámetros del modelo *Tree* o árbol de decisión en Orange.

Utilizaremos una técnica de clasificación llamada bosque aleatorio o *Random Forest* el cual tiene los parámetros especificados en la imagen 5.41.

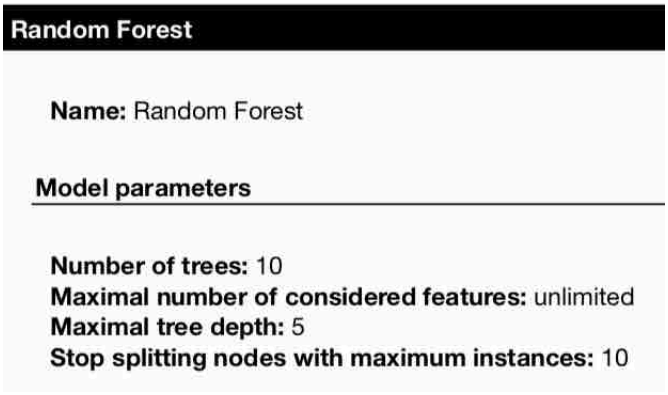


Figura 5.41: Parámetros del modelo *Random Forest* o Bosque aleatorio en Orange.

En el software Orange podemos utilizar la técnica del clasificador bayesiano (*Naive Bayes* dentro del menú de modelos) el cual no requiere parámetros.

La estructura de red neuronal que introduciremos es la que se muestra en la imagen 5.42. Esta estructura es la misma que habíamos descrito en la parte **??**.

La comparación del desempeño de los modelos se muestra en la imagen 5.43.

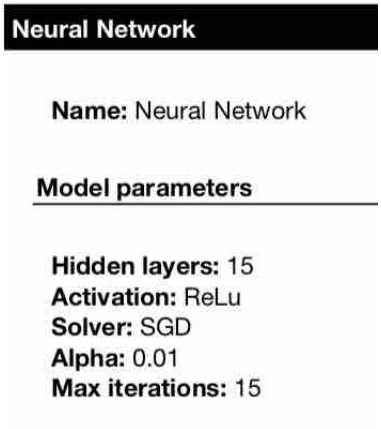
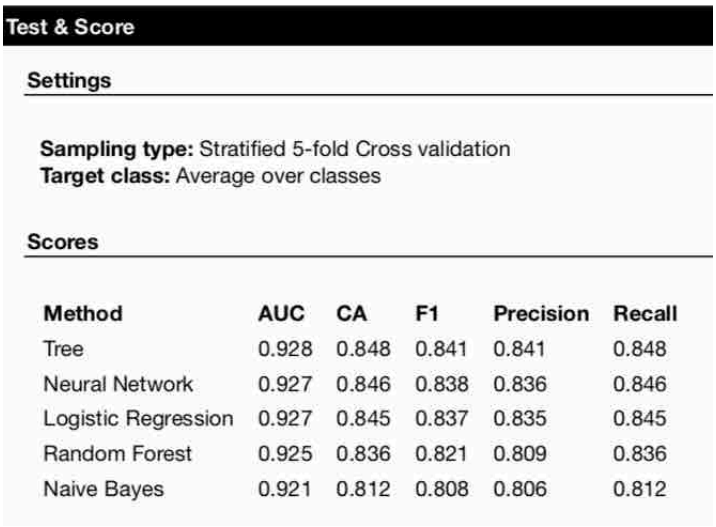


Figura 5.42: Parámetros del modelo *Neural Network* o Red Neuronal en Orange.



Method	AUC	CA	F1	Precision	Recall
Tree	0.928	0.848	0.841	0.841	0.848
Neural Network	0.927	0.846	0.838	0.836	0.846
Logistic Regression	0.927	0.845	0.837	0.835	0.845
Random Forest	0.925	0.836	0.821	0.809	0.836
Naive Bayes	0.921	0.812	0.808	0.806	0.812

Figura 5.43: Comparación entre distintas técnicas ordenadas de acuerdo a su precisión (CA o *Classification Accuracy*) de mayor a menor. Nótese que se ha utilizado la técnica de validación cruzada, estratificada, para evaluar los modelos. Además, para cada medida de desempeño se ha obtenido el promedio en cada clase.

Existen diversos criterios para medir qué tan bueno es un modelo o técnica en la tarea de clasificación, como el área debajo de la curva o *Area Under The Curve* (AUC), la precisión en clasificación o *Classification Accuracy* (CA), valor F o *F-Score* (F1), precisión o *Precision* [3] (Precision en Orange) y sensibilidad o *Recall*(Recall en Orange). En la imagen 5.43 se muestra la comparación del desempeño de los modelos. Nótese que el desempeño de la red neuronal resulta inferior en todas las medidas que ofrece Orange respecto a la técnica de árbol de decisión.

Modificando algunos parámetros en el modelo de la red neuronal 5.44 que

[3]Hemos utilizado la palabra precisión anteriormente en el texto para referirnos más bien a la precisión de clasificación o *CA* en Orange

ofrece Orange obtenemos los resultados que se muestran en la imagen 5.45. Puede observarse que los ajustes le permiten tener mejor desempeño que el resto de los modelos.

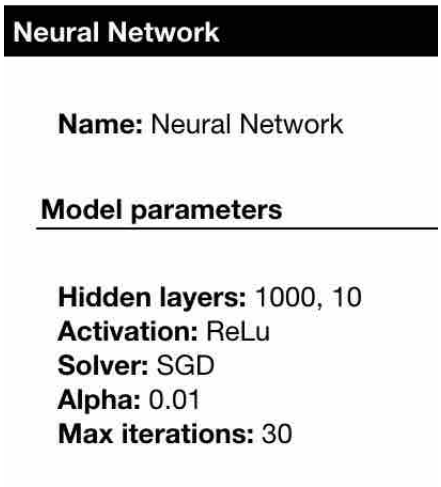
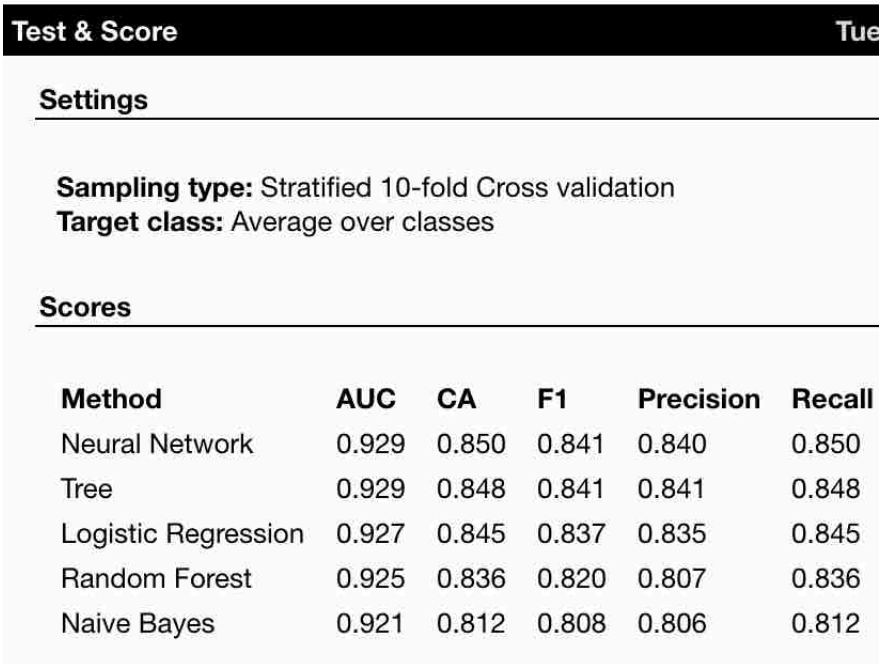


Figura 5.44: La red neuronal con nuevos parámetros. Ahora tenemos una red neuronal con 2 capas ocultas de tamaño 1,000 y 10 respectivamente. El número de épocas se establece en 30.



Method	AUC	CA	F1	Precision	Recall
Neural Network	0.929	0.850	0.841	0.840	0.850
Tree	0.929	0.848	0.841	0.841	0.848
Logistic Regression	0.927	0.845	0.837	0.835	0.845
Random Forest	0.925	0.836	0.820	0.807	0.836
Naive Bayes	0.921	0.812	0.808	0.806	0.812

Figura 5.45: Comparación entre distintos modelos una vez que hemos cambiado la estructura de la red neuronal.

Capítulo 6

Consideraciones finales y Conclusiones

La falta de herramientas tecnológicas que automaticen eficazmente el proceso de gestión de fallas es uno de los grandes problemas reconocidos en la industria de las telecomunicaciones. En este trabajo se abordó el problema de predicción de alarmas de fallas usando redes neuronales artificiales conocidas usualmente como *feed-forward networks* y se corroboró experimentalmente que es posible anticiparse hasta en un 85 % de las fallas presentes en el historial de fallas reales que se usaron en esta tesis.

Los experimentos realizados demuestran que la Red Neuronal arroja resultados satisfactorios en los principales escenarios de fallas identificados en la literatura: predicción de eventos raros, predicción de eventos periódicos y semiperiódicos y predicción de ráfagas (*burst detection*).

Se detallan a continuación algunas conclusiones específicas y/o consideraciones finales.

La técnica de ventanas deslizantes (*sliding window*) es una técnica muy socorrida para abordar el problema de analizar secuencias de datos. En este trabajo, esta técnica jugó un papel importante pues determina la cantidad de información del pasado inmediato (contexto) que la red neuronal utiliza para predecir eventos futuros.

La representación de los datos de entrada que se escogió determina la manera en que la red neuronal percibe el contexto. En trabajos futuros se pueden explorar otras maneras de presentarle la información a una red neuronal o *feed-forward network*.

Derivado también de la manera en que se definió el problema para poder presentárselo a la red neuronal, el problema de predicción pudo entenderse en esta tesis como un problema de clasificación.

La comparación del desempeño de otros métodos de clasificación en el problema de predicción de fallas, nos hizo ver que en algunos escenarios son tan competitivos como la red neuronal.

Apéndice A

Software Orange

Orange es un software de código abierto diseñado por la facultad de informática de la Universidad de Ljubljana para tareas de visualización de datos, aprendizaje autómatico (*Machine Learning*), series de tiempo, minería de datos y otras tareas relacionadas al área de ciencia de datos o estadística [8]. Su principal virtud es poseer un entorno gráfico interactivo amigable para realizar muchas de las tareas anteriores. Se distribuye bajo licencia GPL. Versiones del software superiores a la versión 3.0 incluye componentes en lenguaje C++ con subrutinas escritas en el lenguaje Python.

Orange consiste en una interfaz de Canvas [1] dentro de la cual el usuario puede colocar botones o *widgets* y generar un flujo de trabajo dedicado a analizar los datos, visualizarlos, generar modelos para predicción y clasificación y finalmente evaluar los modelos.

Los principales componentes de Orange son botones llamados *widgets* los cuales permiten llevar a cabo diversas tareas como visualización de datos, preprocesar datos, gráfico de distribuciones, selección de variables, aplicación de modelos, evaluación y predicción de modelos entre otros. Todos los botones tienen parámetros predefinidos que ayudan al usuario a realizar un modelo o cualquier función que el botón indique; no obstante es posible usar Orange como una biblioteca del lenguaje Python, modificar el funcionamiento de un botón o incluso elaborar un script de código en Python, esto último mediante un botón que incluido de manera predeterminada en Orange. Además los componentes o *widgets* están organizados en cajas. Dichas cajas muestran camino estándar a seguir en un proyecto de ciencia de datos, estadística, aprendizaje automático, etc.

A continuación se muestra la organización de los botones o *widgets* en la versión de Orange 3.20.1:

[1] Canvas (o lienzo) es un elemento HTML que permite la creación de gráficos y animaciones de forma dinámica por medio de scripts.

- *Data*: consta los botones cuya función es manipulación de datos; entre ellos figuran la selección de variables, lectura de archivos, muestreo y lectura de bases de datos SQL.
- *Visualize*: como su nombre sugiere, se utilizan para visualizar datos; por ejemplo diagramas de caja, histogramas y gráficos de mosaico.
- *Model*: algoritmos, diversas técnicas o herramientas para predicción. Entre ellos figuran las redes neuronales, la regresión logística y las máquinas de soporte vectorial.
- *Evaluate*: botones destinados a la predicción y a evaluar los modelos o técnicas utilizadas. Entre sus integrantes podemos citar la matriz de confusión, el área debajo de la curva y un botón más robusto de validación de modelos que incluye la técnica de validación cruzada.
- *Unsupervised*: aquí engloba las técnicas de aprendizaje no supervisado; ejemplos de estos botones podemos citar la clusterización y las técnicas de reducción de dimensión como la técnica de componentes principales.

Es posible encontrar herramientas adicionales -en forma de botones o *widgets*- mediante extensiones descargables como el botón de minería de textos, series de tiempo o las regla de asociación.

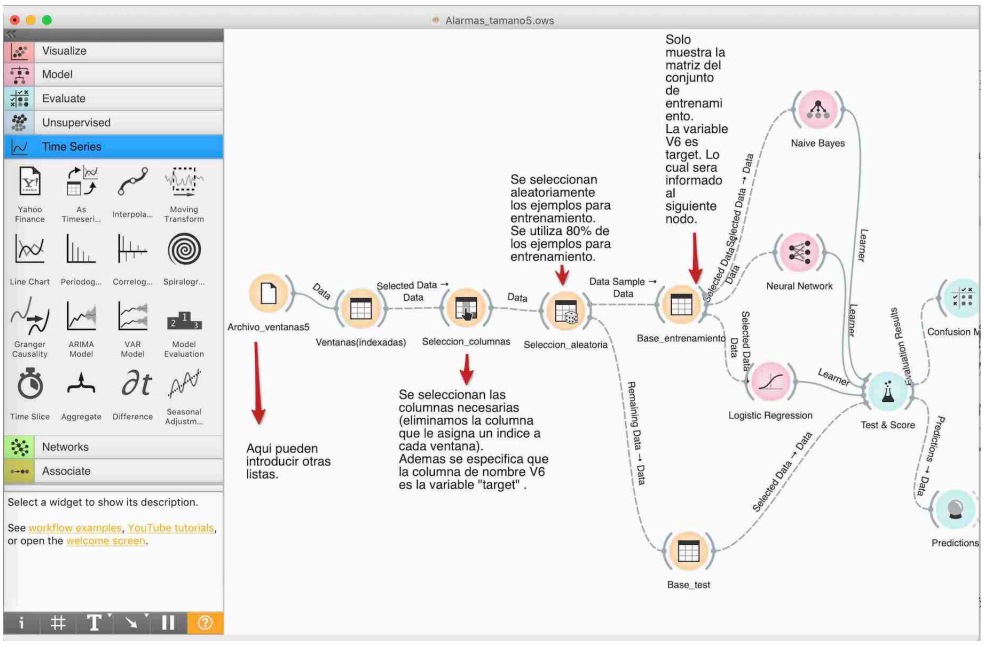


Figura A.1: Interfaz del software Orange en su versión 3.20.1

Apéndice B

Biblioteca Keras

Keras [2] es una biblioteca para redes neuronales de código abierto escrita en Python. Keras provee bloques de construcción de alto nivel para elaborar modelos de aprendizaje automático profundo, dejando tareas de bajo nivel como multiplicación de tensores o convoluciones a bibliotecas especializadas como TensorFlow, Microsoft Cognitive Toolkit o Theano. Keras se incorpora además al lenguaje R mediante una biblioteca de nombre "Keras", lo que permite utilizar las funciones de Keras vía la interfaz de R.

A continuación se muestra el código en R que se utilizó para generar una red neuronal como la empleada en los experimentos con las alarmas reales en el capítulo 5. La lista de alarmas se guarda en una variable llamada alarmas. Esta lista tiene datos categóricos.

Primero cargamos las bibliotecas que vamos a emplear.

```
library(keras)
library(tensorflow)
library(tidyverse)
library(caret)
library (dplyr)
```

Hacemos un histograma de la lista de alarmas.

```
# Usamos el formato Tibble de la biblioteca
# que maneja la biblioteca dplyr para
# hacer un histograma

alarmas1 <- dplyr::as_tibble(alarmas)
colnames(alarmas1) <- c("Valor") # Introducimos
```

```
#nombre a la columna

# Hacemos el histograma de la
# lista de alarmas
alarmas1  %>%
  ggplot(aes(x = Valor , fill= Valor)) +
  geom_bar(width = 1) +
  labs( y= "Numero de alarmas")
```

Creamos la matriz de ventanas. El tamano de la ventana lo guardamos en "tamanoVentana".

```
h <- ( length(alarmas) - tamanoVentana) # Numero de ventanas
#que hay en la lista.

# Matriz cuyos renglones son las ventanas
# de la secuencia.
ventanas <- matrix(nrow =  (length(alarmas)- tamanoVentana) ,
ncol =  tamanoVentana)

# Llenamos la matriz. Cada renglon es una
#ventana de la secuencia.
for(i in 1:h) {
  ventanas[i,] <- alarmas[i:(i+tamanoVentana-1)]
}

# Anadimos etiquetas a la matriz de ventanas.
etiqueta_entrenamiento <- paste("Alarma",
1:(tamanoVentana-1), sep = "")
colnames(ventanas) <- c(etiqueta_entrenamiento, "Target")

view(ventanas)
```

Empezamos a construir la red neuronal especificada en 4.6. Se utiliza como parámetros numAlarmas (número de clases que hay en la lista de alarmas) y epocs (épocas).

```
numVentanas <- nrow(ventanas
)
# Utilizamos la biblioteca Caret para hacer
# muestreo estratificado de las ventanas
# con base en el target
```

```
muestra<- caret::createDataPartition(ventanas[,
tamanoVentana], p=0.7, list=F)

# Dividimos los ejemplos de entrenamiento
ventanas.training <- ventanas[muestra,
1:(tamanoVentana-1)]
ventanas.test <-  ventanas[-muestra,
1:(tamanoVentana-1)]

# Tomamos el target de cada ejemplo
ventanas.trainingtarget <- ventanas[muestra,
tamanoVentana]
ventanas.testtarget <- ventanas[-muestra,
tamanoVentana]


#Convertirmos el target a un vector de enteros
ventanas.trainingtarget <-
as.integer(ventanas.trainingtarget)

ventanas.testtarget <- as.integer(ventanas.testtarget)

#Aplicamos one hot encoding
# One hot encode a los targets de los
# ejemplos de entrenamiento
ventanas.trainLabels <-
keras::to_categorical(ventanas.trainingtarget,
num_classes = numAlarmas)

# One hot encode a los targets de
# los ejemplos test
ventanas.testLabels <-
keras::to_categorical(ventanas.testtarget,
num_classes = numAlarmas)

# Damos un formato adecuado a ciertas variables
# para utilizarlas en algunas funciones
# posteriormente
dimnames(ventanas.training) <- NULL
dimnames(ventanas.trainLabels) <- NULL
ventanas.training <-
as.matrix(ventanas.training)
ventanas.trainLabels <-
as.matrix(ventanas.trainLabels)
```

```
# Anadimos capas al modelo
model %>% keras::layer_dense(units = 1000,
10, activation = 'relu',
input_shape = c(tamanoVentana-1)) %>%
  keras::layer_dense(units = numAlarmas,
  activation = 'softmax')

# Definimos factor de aprendizaje
sgd <- keras::optimizer_sgd(lr = 0.01)

#Podemos incluir criterio de paro
#criterioDeParo <- #keras::callback_early_stopping(
#monitor='val_loss',
#patience=3,
# min_delta = 0.00001,
# mode = "min" )

# Compilamos el modelo
model %>% keras::compile(
  loss = 'categorical_crossentropy', # funcion costo
  optimizer = sgd,  #usamos el gradiente #estocastico
  metrics = 'accuracy'
)

# Corremos el modelo y guardamos el modelo en history
history <- model %>% keras::fit(
  ventanas.training,
  ventanas.trainLabels,
  epochs = epocs,
  suffle= TRUE,
  batch_size = 50,
  validation_split = 0.2
  # callbacks = criterioDeParo
)

# Graficamos el historial del entrenamiento
# plot(history)

# Clasificamos los ejemplos de test con la red entrenada
classes <- model %>% keras::predict_classes( ventanas.test ,
batch_size = 128)

tabla <- table(classes, ventanas.testtarget)
tabla
```

```
# Usamos la funcion evaluacion del modelo de Keras.
score <- model %% keras::evaluate(ventanas.test,
ventanas.testLabels, batch_size = 128)

# Imprimimos el score calculado por la
# biblioteca Keras
print(score)
```

Bibliografía

[1] E.N.S. Camperos and A.Y.A. García. *Redes neuronales: conceptos fundamentales y aplicaciones a control automático*. Automática y robótica. Pearson Educación, 2006.

[2] François Chollet et al. Keras. `https://keras.io`, 2015.

[3] W.G. Cochran. *Sampling Techniques, 3rd Edition*. A Wiley publication in applied statistics. Wiley India Pvt. Limited, 2007.

[4] Colaborades de Wikipedia. Overfitting — Wikipedia, the free encyclopedia, 2004. [Consulta en la versión en inglés 09-Febrero-2020].

[5] Colaborades de Wikipedia. Rectifier (neural networks) — Wikipedia, the free encyclopedia, 2004. [Consulta en la versión en inglés 09-Febrero-2020].

[6] Colaborades de Wikipedia. Sigmoid function — Wikipedia, the free encyclopedia, 2004. [Consulta en la versión en inglés 09-Febrero-2020].

[7] Felipe Correa, José Gallardo, Nelson Muñóz, and Ricardo Pérez. Estudio comparativo basado en métricas para diferentes arquitecturas de navegación reactiva. *Ingeniare. Revista chilena de ingeniería*, 24:46 – 54, 01 2016.

[8] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan. Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, 14:2349–2353, 2013.

[9] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[10] J. Feldman and R. Rojas. *Neural Networks: A Systematic Introduction*. Springer Berlin Heidelberg, 2013.

[11] Tape Thomas G. The area under a ROC curve, 2019. Último acceso 14 de Abril de 2019.

[12] Alexis Gabadinho, Gilbert Ritschard, Matthias Studer, and Nicalas S Müller. Mining sequence data in R with the TraMineR package: A users guide for version 1.2. 2009.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] Minitab Inc. ¿Qué son los errores de tipo I y tipo II? `https://support.minitab.com/es-mx/minitab/18/help-and-how-to/statistics/basic-statistics/supporting-topics/basics/type-i-and-type-ii-error`, 2019. [Último acceso 22-Diciembre-2019].

[15] Mohammad Jaudet, Nacem Iqbal, and Amir Hussain. Neural networks for fault-prediction in a telecommunications network. In *8th International Multitopic Conference, 2004. Proceedings of INMIC 2004.*, pages 315–320. IEEE, 2004.

[16] Sánchez Bojórquez L. *Sistemas inteligentes y su aplicación a gestión de fallas en redes complejas de telecomunicación*. Tesisina de licenciatura, UNAM, 2013.

[17] Alberto Leon-Garcia and Indra Widjaja. *Communication networks*. McGraw-Hill, Inc., 2003.

[18] Tao Li, Chunqiu Zeng, Yexi Jiang, Wubai Zhou, Liang Tang, Zheng Liu, and Yue Huang. Data-driven techniques in computing system management. *ACM Computing Surveys (CSUR)*, 50(3):45, 2017.

[19] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, Sep 1997.

[20] Michael Nielsen. If correlation doesn't imply causation, then what does? `http://www.michaelnielsen.org/ddi/if-correlation-doesnt-imply-causation-then-what-does`, June 2018.

[21] Michael Nielsen. Improving the way neural networks learn. `http://neuralnetworksanddeeplearning.com/chap3.html`, June 2018.

[22] Judea Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, New York, NY, USA, 2nd edition, 2009.