Grado en Matemáticas

TRABAJO FIN DE GRADO

Deep Learning: Redes Neuronales Convolucionales en R

Inmaculada Aguado López

Sevilla, septiembre de 2020

Índice general

Resumen V
Abstract VI
Índice de Figuras VII

1. Introducción 1
1.1. Inteligencia Artificial 1
1.1.1. Historia 2
1.1.2. Enfoques de la Inteligencia Artificial 3
1.1.3. Limitaciones de la Inteligencia Artificial 4
1.2. Machine Learning 4
1.2.1. Tipos de Aprendizaje Automático 4
1.2.1.1. Aprendizaje Supervisado 5
1.2.1.2. Aprendizaje No supervisado 5
1.3. Deep Learning 6

2. Redes Neuronales 7
2.1. Inspiración biológica 7
2.2. Redes Neuronales Artificales 8
2.2.1. Definición 8
2.2.2. Arquitectura de las RNA 8
2.2.3. Elementos de una RNA 9
2.3. Perceptrón 10
2.3.1. Algoritmo de convergencia del perceptrón 12
2.4. Redes multicapas 15
2.4.1. Propagación de los patrones de entrada 16
2.4.2. Algoritmo de Retropropagación 17
2.4.2.1. Regla delta generalizada 18
2.4.2.2. Proceso de aprendizaje de una red multicapa 22
2.4.2.3. Early stopping 22

3. Regularización del Deep Learning 25
3.1. El error de generalización 25
3.2. El problema del sobreaprendizaje 26
3.3. Regularización 27
3.3.1. Regularización de la funcion de coste 28

4. Redes Neuronales Convolucionales (CNN) 29
4.1. Introducción 29
4.2. La operación de convolución 30

4.3. Capas convolucionales . . . 32
4.4. Capas de Pooling . . . 33
4.5. Capas full-conected . . . 34
4.6. Capas de softmax y de clasificacion . . . 35

5. La visión artifical y las CNNs **37**
5.1. Preprocesamiento . . . 37
5.2. Normalización del contraste . . . 38
5.3. Aumento del conjunto de datos de entrada . . . 39
5.4. Aplicación . . . 40
5.4.1. Set de datos . . . 40
5.4.2. Aplicación en R . . . 40
5.4.2.1. Introducción a R . . . 40
5.4.2.2. Librerías utilizadas . . . 41
5.4.2.3. Aplicación . . . 42

Bibliografía **49**

A mi madre, mi hermano y a toda persona de mi familia
o amigos que en algún momento ha estado ahí.
Gracias por apoyarme durante mi etapa académica que hoy acaba, de momento.

Resumen

En los últimos años se ha producido un gran auge en el campo de la Inteligencia Artificial al que hemos asistido y este ha sido posible en gran parte gracias a las Redes Neuronales. El Deep Learning ha conseguido que estas redes se especialicen en detectar determinandas características ocultas de los datos. Esto ha sido posible gracias a que usa una metodología de aprendizaje análoga a la que usan los seres humanos para aprender. Numerosos campos de investigación se han beneficio de este auge, como la visión artificial.

En este proyecto, tras introducir el concepto de red neuronal y presentar la metodología que se seguirá, se hará uso de estas redes neuronales y en concreto de una red neuronal convolucional para desarrollar un algoritmo en el lenguaje R que sea capaz de reconocer caracteres escritor a mano de una extensa base de datos, concretamente se usarán números del 0 al 9. Se busca como objetivo conseguir al menos una precisión del 95 % en la clasificación de los mismos.

// Abstract

In recent years there has been a great boom in the field of Artificial Intelligence that we have attended and this has been made possible in large part thanks to the Neural Networks. Deep Learning has achieved that these networks specialize in detecting certain hidden characteristics of the data. This has been possible because it uses a learning methodology similar to that used by humans to learn. Numerous fields of research have benefited from this boom, such as artificial vision.

In this project, after introducing the concept of neural network and presenting the methodology that will be followed, these neural networks will be used and in particular a convolutional neural network to develop an algorithm in the R language that is capable of recognizing writer-to-character characters. hand of an extensive database, specifically numbers from 0 to 9 will be used. The objective is to achieve at least 95 % accuracy in their classification.

Índice de figuras

1.1. Campos de la Inteligencia Artificial 1
1.2. 1950-1970. Redes Neuronales 2
1.3. 180-2010. Aprendizaje automatico 3
1.4. Día presente. Deep Learning 3

2.1. Estructura básica de una neurona 7
2.2. Arquitectura de una RNA 9
2.3. Representación gráfica de un perceptrón de una sola neurona. 11
2.4. Modelo de un perceptrón simple. 11
2.5. Regiones de decisión de un problema de clasificación de patrones de dos clases. 12
2.6. Modelo de un perceptrón simple con una entrada fija. 13
2.7. Representación del criterio de early-stopping 23

4.1. Estructura clásica de una red convolucional para clasificación 30
4.2. Ejemplo de operación de convolución 31
4.3. Ejemplo de reducción de muestreo 34
4.4. Ejemplo de capa totalmente conectada 35

5.1. Macroarquitectura de la red VGG16 37
5.2. Comparación normalización de contraste global y local 39
5.3. Ejemplo del conjunto de datos 40
5.4. Primer número del conjunto de datos 43

Capítulo 1

Introducción

1.1. Inteligencia Artificial

La Inteligencia Artificial (IA) junto con la experiencia permiten a las máquinas aprender, mejorar con nuevos datos que se le aporten y además es capaz de realizar tareas que en el pasado solo eran capaces de hacer los humanos. Actualmente podemos escuchar la palabra Inteligencia Artificial en diferentes ámbitos y algunos de los ejemplos más frecuentes son los ordenadores que juegan al ajedrez o los coches que se conducen solos. Estos avances que permiten cada vez más complejidad son posibles sobre todo al aprendizaje profundo (Deep Learning). El objetivo de estas tecnologías es entrenar máquinas para que mediante un procesamiento de datos y reconocimiento de patrones lleven a cabo las tareas específicas que se les indiquen.

Una definición concisa acerca del campo de la IA sería: *el esfuerzo de automatizar tareas intelectuales normalmente realizadas por humanos*. La Inteligencia Artificial engloba diferentes campos como son Machine Learning, que a su vez engloba al Deep Learning, estadística, razonamiento lógico o algoritmos de búsqueda.

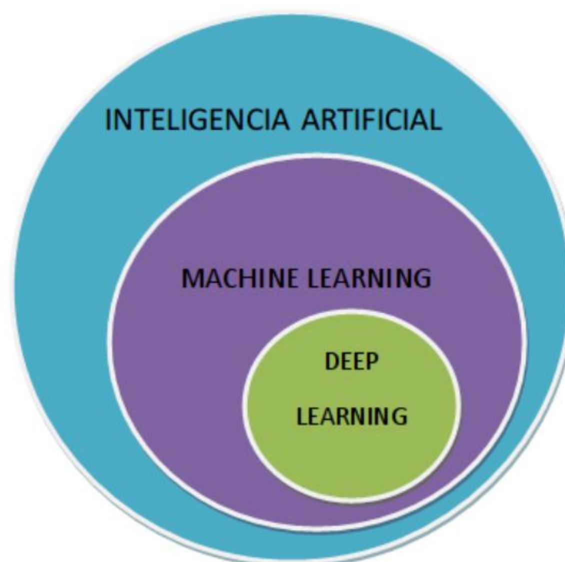


Figura 1.1: Campos de la Inteligencia Artificial

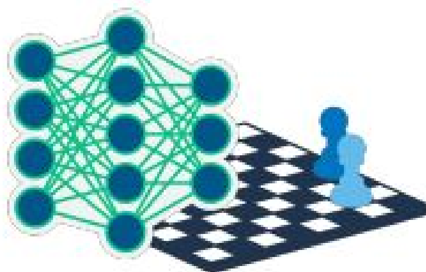
1.1.1. Historia

En 1956 John McCarthy, Marvin Minsky y Claude E.Shannon acuñaron el término de Inteligencia Artificial en la Conferencia de Dartmouth, conferencia que ellos mismos junto a Nathaniel Rochester organizaron. En esa conferencia predijeron que en aproximadamente 10 años la sociedad entera estaría rodeada de máquinas inteligentes, algo que ahora sabemos que estaba lejos de la realidad. La edad de oro de la IA se ha establecido a finales de los años 90. A partir de entonces las películas de Hollywood y las novelas de ciencia ficción siempre relacionaban la IA con robots capaces de replicar el comportamiento de los seres humanos para así ganar la lucha por la conquista de nuestro mundo. La realidad es que la Inteligencia Artificial se usa para obtener más beneficios en las industrias.

En los últimos tiempos el término IA cada vez ha ido siendo más popular y el número de personas que la han usado también ha aumentado. Como consecuencia se ha incrementado el volumen de datos usados y se han desarrollado continuamente algoritmos mucho más complejos y avanzados, también gracias a las mejores en el cómputo y almacenamiento de los datos.

En la siguiente secuencia de imágenes se resume cómo ha evolucionado la IA desde sus comienzos:

- Las primeras pruebas con redes neuronales hace despertar la emoción por las llamadas "máquinas pensantes"



1950-1970

Redes neurales

Figura 1.2: 1950-1970. Redes Neuronales

- Se vuelve popular el Aprendizaje Automático (Machine Learning).

Figura 1.3: 180-2010. Aprendizaje automatico

- Los adelantos en Aprendizaje Profundo (Deep Learning) impulsan el auge de la Inteligencia Artificial.

Figura 1.4: Día presente. Deep Learning

1.1.2. Enfoques de la Inteligencia Artificial

La Inteligencia Artificial es un campo que abarca diferentes enfoques:

- **Máquinas que intentan imitar el comportamiento de los humanos.** La finalidad aquí es ser capaz de desarrollar una máquina que pueda realizar trabajos específicos que antes eran exclusivos de seres humanos inteligentes. Un ejemplo de lo que se puede llegar a conseguir es la famosa Prueba de Turing. Para esta prueba sería necesario una máquina que tuviera todas estas capacidades humanas:
    - Procesamiento del lenguaje natural, para comunicarse.
    - Representación del conocimiento que le permita almacenar todo aquello que va conociendo.

   - Razonamiento automático, para que pueda usar todo lo que tiene almacenado y consiga responder preguntas.
   - Aprendizaje automático, para detectar patrones y nuevas circunstancias.
   - Visión computacional que le permita percibir objetos.
   - Robótica, para manipular y mover objetos.

- **Máquinas que intentan imitar los pensamientos humanos.** En este caso el objetivo consiste en desarrollar máquinas que sean capaces de pensar literalmente como lo hace un ser humano. Esto hace que la máquina desarrollada posea capacidades cognitivas como puede ser la toma de decisiones o la resolución de problemas.
- **Máquinas que piensan de manera racional.** El objetivo es hacer que la máquina piense de manera racional. La Lógica es el ejemplo más claro de este enfoque.
- **Máquinas que se comportan de manera racional** El fin de este enfoque es conseguir que la máquina actúe de tal forma que el resultado obtenido será el más óptimo.

1.1.3. Limitaciones de la Inteligencia Artificial

A pesar de que la Inteligencia Artificial ha introducido importantes avances a nivel industrial, esta posee limitaciones que pueden afectar al objetivo final que se plantea.

La IA está totalmente condicionad con la cantidad de datos que tenemos, ya que es la manera que tienen las máquinas de aprender. Cuando tenemos unos datos cuya calidad no es la adecuada los resultados que se obtiene no se acercan a los esperados. Por otra parte las máquinas de IA se entrenan para que realicen una tarea concreta definida anteriormente. Es decir, si entrenas a una máquina para que juegue al ajedrez no va a ser capaz de jugar a otro juego de mesa. En este aspecto estás muy lejos del comportamiento de los seres humanos

1.2. Machine Learning

Dentro de la Inteligencia Artificial se puede hablar del Aprendizaje Automático o Machine learning como una de las dos ramas más populares. Del mismo modo no es tarea sencilla definir qué significa. Una definición podría ser la capacidad que posee una máquina para aprender sin que se haya programado con anterioridad.

El objetivo que tiene el Machine Leatning es conseguir unas reglas de comportamiento a partir de unos datos reales donde se aplican esas reglas. Eso lo consigue gracias a la combinación de distingas técnicas como las matemáticas, las estadísticas o las ciencias de la computación.

1.2.1. Tipos de Aprendizaje Automático

Existen diversas técnicas de Aprendizaje Automático, si nos fijamos en la filosofía usada para obtener el conocimiento de los datos podemos clasificarlas en dos tipos:

1.2.1.1. Aprendizaje Supervisado

En este tipo de aprendizaje los modelos se entrenan con un conjunto de datos en los que los resultados de salida se conocen previamente. Los máquinas aprenden a través del ajuste de unos parámetros y estableciendo reglas cuales son estos resultados. Cuando se tengan nuevos datos sin etiquetar, el algoritmo ya entrenado será capaz de predecir el resultado deseado.

Hay dos aplicaciones principales de aprendizaje supervisado: Clasificación y Regresión.

- Clasificación

  El objetivo de este tipo de aprendizajes es predecir las clases o grupos categóricos a los que pertenece cada dato. En este caso son valores discretos.

- Regresión

  En este caso el objetivo es un resultado contínuo. A partir de un número de variables predictoras (explicativas) y una variable respuesta se encuentran las relaciones entre dichas variables y se entrena un modelo capaz de predecir la variable respuesta con la mayor precisión posible.

1.2.1.2. Aprendizaje No supervisado

A diferencia del aprendizaje supervisado, en el aprendizaje no supervisado tenemos unos datos cuya pertenencia a un grupo está sin etiquetar. El objetivo de estos modelos será conseguir una estructura de los datos según su comportamiento, distancia en el espacio, etc.

Se pueden distinguir dos categorías: Agrupamiento o Clusterización y Reducción Dimensional.

- Agrupamiento o Clusterización.

  La Clusterización es una técnica de Machine Learning usada para agrupar información según los datos de entrenamiento usados sin que la máquina sepa previamente cómo se deben estructurar Cada grupo en los que se diferencias los datos son similares entre si y poseen diferencias con respecto al resto de grupos. Un elemento perteneciente a un clúster concreto, pertenece a esa por una serie de condiciones que lo hacen apto para ello.

- Reducción dimensional

  A la hora de entrenar una máquina puede darse la situación de que el número de características que se tengan de unos datos sea excesivamente alto, es decir, una alta dimensionalidad. Esto implica una dificultad en la capacidad de procesamiento, por eso es importante reducir dicha dimensionalidad y así disminuir el problema encontrado. La manera mas habitual de hacerlo es identificando correlaciones entre las características para asi eliminar aquellas que aporten la misma información o a traces de combinaciones entre más de una característica obtener una sola que sea capaz de dar casi la totalidad de la información que daban todas por separado. Finalmente, lo que se obtiene es un subconjunto de datos con la mayoría de información.

1.3. Deep Learning

La otra rama que vamos a ver dentro de la IA es el Aprendizaje Profundo o Deep Learning. Lo que caracteriza este subcampo es la utilización de diferentes estructuras de redes neuronales que permiten un aprendizaje mucho más significativo sobre los datos. El *profundo* o *deep* hace referencia a la cantidad de capas de representación que se usan en el modelo.

La primera capa de cualquier red neuronal toma como entrada los datos en bruto, es decir, el conjunto de datos del que partimos sin ningún procesamiento. A continuación se procesan y se extrae la información correspondiente y la pasa a la siguiente capa. Es un proceso iterativo que acaba cuando los datos llegan a la final, que es la llamada de predicción, ya que da como resultado la predicción buscada. Una vez obtenida esta predicción se compara con el dato original y así la red neuronal puede ir ajustando sus hiperparámetros para obtener mejores resultados. Es uno de los principales algoritmos usados en programas para reconocimiento de imágenes.

Lo que caracteriza de manera singular a las redes neuronales artificiales es la capacidad de resolver una serie de problemas que muy dificilmente un ser humano hubiera resuelto con el único soporte de un ordenador.

Casi todos los algoritmos de *deep learning* se pueden resumir en cumplir los pasos de la siguiente receta básica:

- Recopilar un conjunto de datos asociados al problema.
- Diseñar una función de coste apropiada al problema, también conocida como función de pérdida [*loss function*]
- Seleccionar un modelo de red neuronal y establecer sus hiperparámetros (tamaño, características,...)
- Aplicar un algoritmo de optimización para minimizar la función de coste ajustando los parámetros de la red.

Capítulo 2

Redes Neuronales

2.1. Inspiración biológica

El sistema nervioso es una red compleja que permite la realización de todas las funciones del organismo. En el ámbito de la informática y las telecomunicaciones se suele comparar con un ordenador. Esta comparación se debe a las analogías encontradas en su funcionamiento y estructura. Las unidades periféricas (órganos) transmiten una gran cantidad de información a través de los cables (nervios). La unidad de procesamiento central (cerebro), donde se encuentran almacenados estos datos (memoria), los ordena, analiza, etc.

La unidad básica del sistema nervioso es la neurona, formada por una parte central que llamamos cuerpo y unas prolongaciones. Dentro de estas prolongaciones se encuentras las dendritas, utilizadas como entradas y el axón, utilizado como salida. Por último, encontramos la sinapsis que conecta el axón de una neurona con las dendritas de otras neuronas.

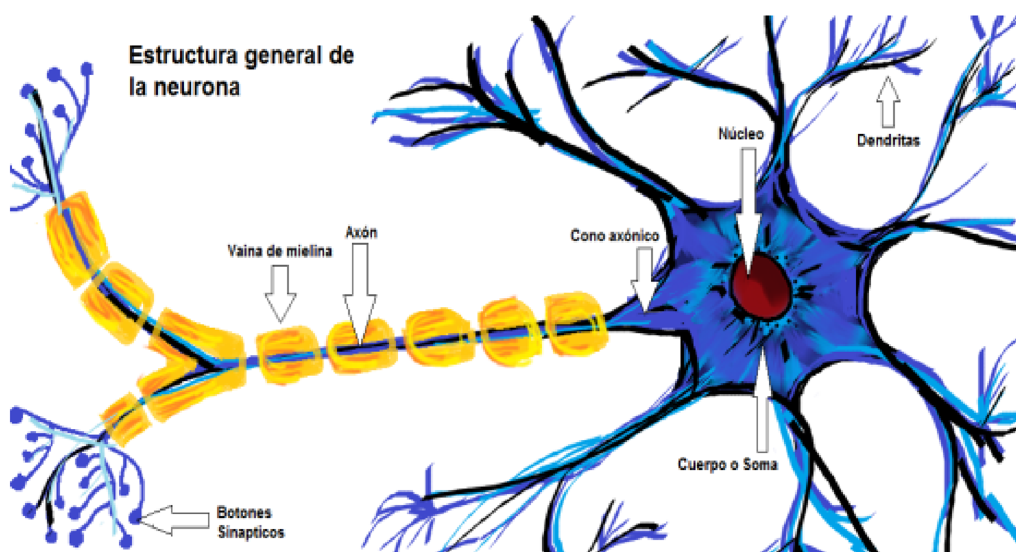


Figura 2.1: Estructura básica de una neurona

Las Redes Neuronales Artificiales son dispositivos programados para imitar el funcionamiento de las redes neuronales biológicas de los seres vivos. Estas redes se caracterizan por la capacidad de aprendizaje. Algunos problemas del mundo real tales como la asociación, evaluación o el reconocimiento de patrones se resuelven mejor con las redes neuronales así como los seres humanos son también los que mejor resuelven este tipo de problemas. Cuando no existe un algoritmo definido claramente para la resolución de un problema es donde son perfectas las redes neuronales.

2.2. Redes Neuronales Artificales

2.2.1. Definición

Las Redes Neuronales Artificiales (RNA) como hemos definido anteriormente son un modelo computacional inspirado en el modelo que componen las redes neuronales biológicas. Consiste en un conjunto de unidades, neuronas artificiales que están conectadas entre si y que permiten la transmisión de señales. Estas señales atraviesan la red neuronal donde se someten a diferentes operaciones y donde finalmente dan como resultado unos valores de salida.

2.2.2. Arquitectura de las RNA

La arquitectura de una RNA es la estructura que tienen las conexiones de la red. Estas conexiones, sinápticas, son direccionales, es decir, la información solo va a se transmitida en un sentido.

Las neuronas están agrupadas en unidades que llamamos capas. Es posible realizar una clasificación de estas capas:

- Capa de entrada: se encargan de la recepción de los datos que proceden del exterior.
- Capas ocultas: estas capas no reciben ni transmiten información, se encargan de procesar internamente los datos.
- Capa de salida: proporciona la respuesta a las señales de la entrada.

Generalmente, las conexiones son entre neuronas pertenecientes a diferentes capas aunque son posibles las conexiones intercapas o de realimentación, aquellas que no siguen el sentido de entrada-salida.

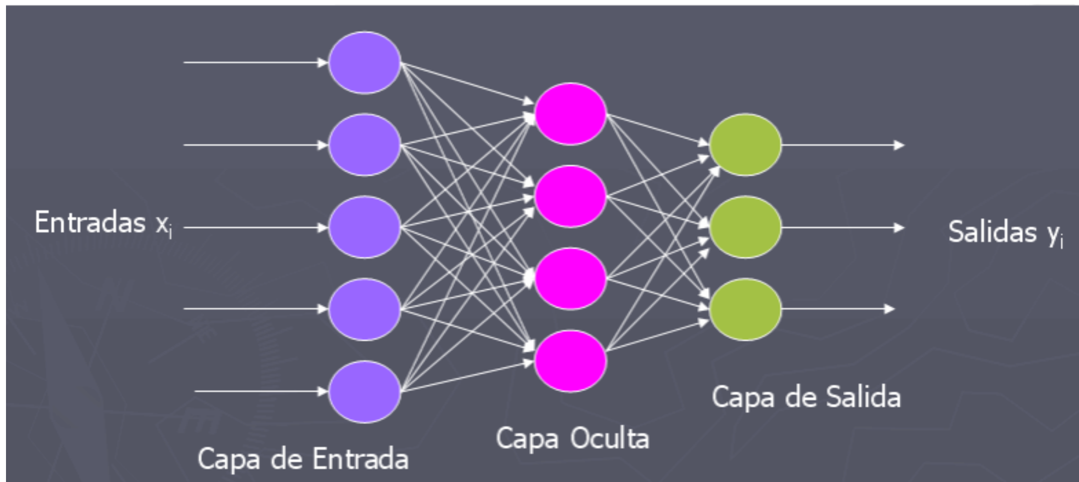


Figura 2.2: Arquitectura de una RNA

2.2.3. Elementos de una RNA

En una neurona artificial se pueden distinguir los siguiente elementos:

- Un *conjunto de entradas* x_j.
- Conjunto de pesos sinápticos w_{ij}, con $j = 1, ..., n$. Representan la interacción entre la neurona presináptica j y la postsináptica i.
- Una *regla de propagación* h_i, definida a partir del conjunto de entradas y los pesos sináptico. Es decir:

  $$h_i(x_1, \ldots, x_n, w_{i_1}, \ldots, w_{i_n})$$

  La regla de propagación que normalmente se usa es la combinación lineal de las entradas y los pesos, obteniéndose:

  $$h_i(x_1, \ldots, x_n, w_{i_1}, \ldots, w_{i_n}) = \sum_{i=1}^{n} w_{ij} x_j$$

  Además se suele añadir otro parámetro, θ_i, denominado umbral, que se resta al potencial postsináptico.

  $$h_i(x_1, \ldots, x_n, w_{i_1}, \ldots, w_{i_n}) = \sum_{i=1}^{n} w_{ij} x_j - \theta_i$$

- Una *función de activación*, la cual representa simuntáneamente la salida de la neurona y su estado de activación. Normalmente los valores de la salida estan comprendidos en un rango (0,1) o (-1,1). Si denotamos por y_i dicha función de activación, obtenemos:

  $$y_i = f_i(h_i) = f_i\left(\sum_{j=0}^{n} w_{ij} x_j\right)$$

  De todas las funciones de activación que existen siempre se intentará buscar aquella cuya derivada sea simple ya que esto hará que se minimice el coste computacional. Algunas de estas funciones de activación más utilizadas son:

- Sigmoid-Sigmoide.

  La función sigmoide transforma los valores que se introduzcan a una escala (0,1), donde aquellos valores altos tienden de manera asintótica al valor 1 mientras que los valores muy bajos lo hacen al valor 0.

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

- Tanh: Tangente hiperbólica.

  La función tangente hiperbólica realiza la misma tranformación anterior pero a una escala (-1,1).

$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

- Rectified Lineal Unit.

  La función ReLU consiste en igualar a 0 los valores negativos y dejar los positivos tal y como están.

$$f(x) = max(0, x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } x >= 0 \end{cases}$$

- Leaky Rectified Lineal Unit

  La función Leaky ReLU tiene la particularidad de que a diferencia de la función ReLU simple, esta multiplica los valores negativos por un coeficiente rectificativo.

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

- Softmax.

  La función Softmax trandforma las salidas a una representación en forma de probabilidades, de tal manera que el sumatorio de todas las probabilidades de las salidas de 1.

$$f(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z}}$$

2.3. Perceptrón

El perceptrón simple es la forma más simple que existe de red neuronal. Estos son usados para la clasificación ya que son los más eficaces en la identificación de patrones. En el sentido del aprendizaje automático es un clasificador binario por lo que será capaz de decidir si una entrada pertenece a una clase u otra. , es un clasificador lineal ya que los patrones deben ser linealmente separables, es decir, los patrones se encuentran a ambos lados de un hiperplano. De manera simple se puede decir que un perceptrón es una neurona con pesos sinápticos y umbral ajustables.

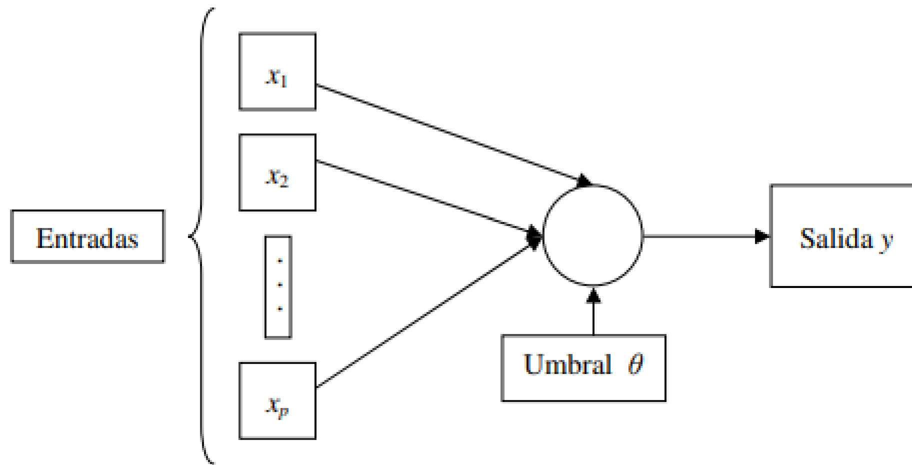


Figura 2.3: Representación gráfica de un perceptrón de una sola neurona.

El modelo de una neurona artificial está compuesto por un combinador lineal al que le sigue un limitador. Un ejemplo de este tipo de modelo se puede observar en la siguiente figura. La suma de los nodos mide una combinación lineal de cada una de las entradas,x_i aplicadas a sus sinápsis,w_i. A esta suma se le aplica un limitador θ y como consecuencia la neurona da como resultado una salida igual a 1 si la entrada del limitador es positiva o -1 si es negativa.

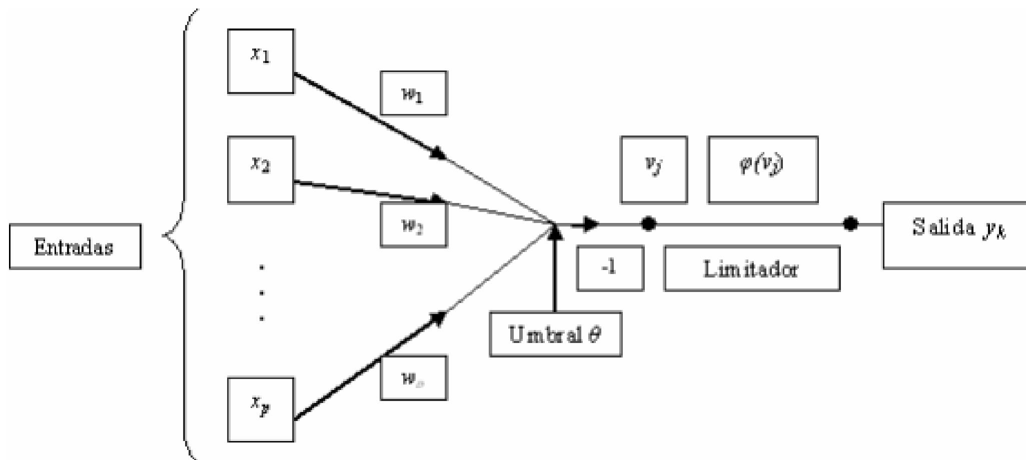


Figura 2.4: Modelo de un perceptrón simple.

En la figura anterior los pesos sinápticos son denotados por $w_1, w_2, ..., w_p$ y las entradas aplicadas al perceptrón por $x_1, x_2, ..., x_p$. El umbral es θ. La salida del combinador lineal (entrada del limitador) es :

$$v = \sum_{i=1}^{p} w_i x_i - \theta$$

El objetivo del perceptrón como ya hemos comentado será clasificar el conjunto de entrada en dos clases, llamadas ζ_1 y ζ_2. Para saber a que clase pertenecen se mirará la salida del perceptrón. Si esta es positiva pertenecerá a la clase ζ_1 y en caso contrario a la clase ζ_2.

La dos regiones de decisión están separadas por un hiperplano, definido por :

$$\sum_{i=1}^{p} w_i x_i - \theta = 0$$

Para ver una representación gráfica consideremos el caso de dos variables de entrada x_1 y x_2. Un punto (x_1, x_2) que se encuentre por encima de la línea que divide las dos regiones se asigna a la clase ζ_1 y un punto que se encuentre por debajo se asigna a la clase ζ_2. El efecto que tiene el umbral es desplazar la región de decisión del origen.

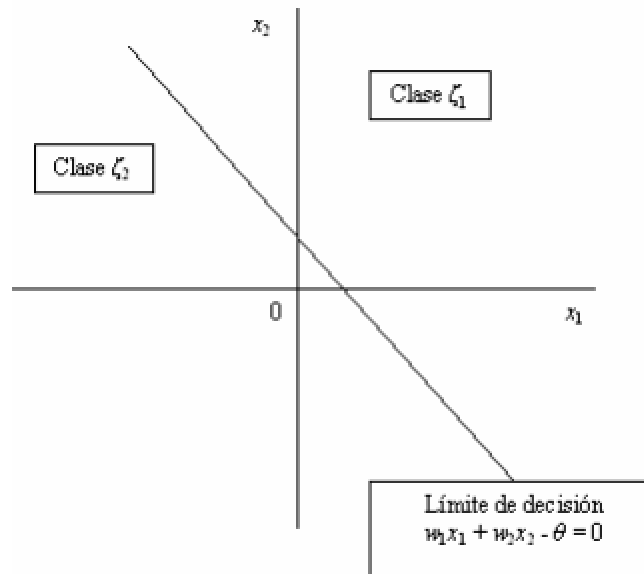


Figura 2.5: Regiones de decisión de un problema de clasificación de patrones de dos clases.

2.3.1. Algoritmo de convergencia del perceptrón

Los pesos sinápticos $w_1, w_2, ..., w_p$ del perceptrón pueden ser fijados o adaptados mediante un algoritmo conocido como algoritmo de convergencia del perceptrón donde se usa una regla de corrección de error. Para desarrollar dicho algoritmo de aprendizaje vamos a considerar el modelo descrito en la siguiente figura.

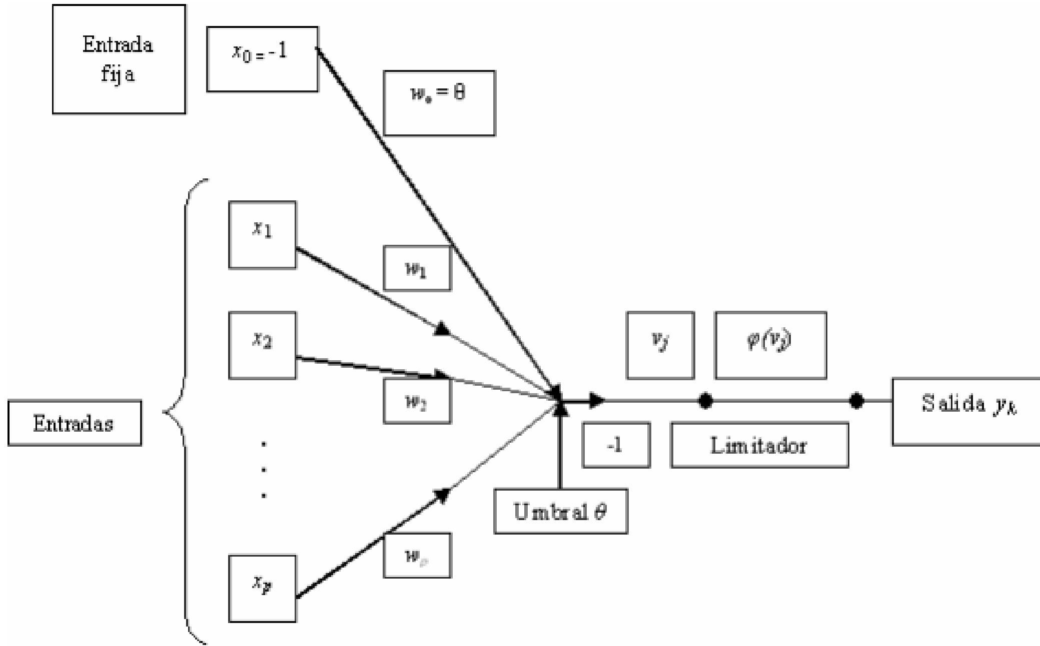


Figura 2.6: Modelo de un perceptrón simple con una entrada fija.

En este caso el umbral θ_n se trata como un peso sináptico conectado a una entrada fija e igual a -1. Podemos definir el vector de entrada como:

$$x(n) = [-1, x_1(n), x_2(n), ..., x_p(n)]^T$$

Del mismo modo definimos el vector de pesos como:

$$w(n) = [\theta(n), w_1(n), w_2(n), ..., w_p(n)]^T$$

La salida del combinador lineal puede ser escrita de forma compacta como se muestra en la siguiente ecuación:

$$v(n) = w^T(n)x(n)$$

Ahora bien, dado un n fijado, mediante la ecuación $w^T x = 0$ se define el hiperplano como superficie de decisión entre las dos clases diferentes de entradas.

Suponemos, de esta manera que las variables de entrada del perceptrón de una capa originan dos clases linealmente separables a ambos lados de un hiperplano. Denotaremos por χ_1 al subconjunto de vectores de entrada $x_1(1), x_1(2), ...$ que pertenecen a la clase ζ_1, y por χ_2 al subconjunto $x_2(1), x_2(2), ...$ que pertenecen a la clase ζ_2. La unión de χ_1 y χ_2 da como resultado el vector completo de entrada χ.

Dados los subconjuntos de entrenamiento χ_1 y χ_2, el problema de entrenamiento del perceptrón es encontrar un vector de pesos w que cumpla las dos inecuaciones siguientes:

$$w^T(n)x(n) \geqslant 0 \quad \text{para cada vector de entrada } x \text{ que pertenezca a la clase } \zeta_1.$$

y

$$w^T(n)x(n) < 0 \quad \text{para cada vector de entrada } x \text{ que pertenezca a la clase } \zeta_2.$$

El algoritmo para ajustar el vector de pesos se puede formular como:

1. Si el enésimo elemento del vector de entrenamiento $x(n)$ es clasificado correctamente por el vector de pesos $w(n)$ calculado en la enésima iteración del algoritmo, no se lleva a cabo ninguna corrección del vector de pesos del perceptrón, como se muestra a continuación:

$$w(n+1) = w(n) \quad \text{si} \quad w^T(n)x(n) \geqslant 0 \quad \text{y} \quad x(n) \quad \text{pertenece a la clase} \quad \zeta_1.$$

y

$$w(n+1) = w(n) \quad \text{si} \quad w^T(n)x(n) < 0 \quad \text{y} \quad x(n) \quad \text{pertenece a la clase} \quad \zeta_2.$$

2. En caso contrario, el vector de pesos se actualiza usando la siguiente regla:

$$w(n+1) = w(n) - \eta(n)x(n) \quad \text{si} \quad w^T(n)x(n) \geqslant 0 \quad \text{y} \quad x(n) \quad \text{pertenece a la clase} \quad \zeta_1.$$

y

$$w(n+1) = w(n) + \eta(n)x(n) \quad \text{si} \quad w^T(n)x(n) < 0 \quad \text{y} \quad x(n) \quad \text{pertenece a la clase} \quad \zeta_2.$$

donde el parámetro tasa de aprendizaje $\eta(n)$ controla el ajuste aplicado al vector de pesos en el iteracion n.

.

El *teorema de convergencia de incremento finito* para un perceptrón de una capa afirma que :

Sean los subconjuntos de entrenamiento χ_1 y χ_2 linealmente separables. Sean las entradas presentadas al perceptrón las originadas por esos subconjuntos. El perceptrón converge tras un número finito de iteraciones, n_0, en el sentido de que

$$w(n_0) = w(n_{0+1}) = w(n_{0+2}) = ...$$

es un vector de soluciones para $n_0 \leqslant n_{max}$.

Ahora bien, definamos los pasos que componen dicho algoritmo:

- *Variables y parámetros*

  $x(n) = [-1, x_1(n), x_2(n), ..., x_p(n)]^T$ es el vector de entradas.

  $w(n) = [\theta(n), w_1(n), w_2(n), ..., w_p(n)]^T$ es el vector de pesos.

  $\theta(n)$ es el umbral.

  $y(n)$ es la respuesta actual.

  $d(n)$ es la respuesta deseada.

  η es el parámetro tasa de aprendizaje, una constante positiva menor que la unidad.

- *Paso 1:* Inicialización.

  Hacemos $w(0) = 0$ y se llevan a cabo los diferentes cálculos en los instantes de tiempo $n = 1, 2, ...$

- *Paso 2:* Activación

  En el instante de tiempo n, se activa el perceptrón aplicando el vector de entrada $x(n)$ y la respuesta deseada $d(n)$.

- *Paso 3:* Cálculo de la respuesta actual.

  Se calcula la respuesta actual del perceptrón

  $$y(n) = sgn[w^T(n)x(n)]$$

  donde $sgn(n)$ es la función del signo (vale 1 si $n > 0$, y -1 si $n < 0$).

- *Paso 4:* Adaptación del vector de pesos.

  Se actualiza el vector de pesos del perceptrón:

  $$w(n+1) = w(n) + \eta[d(n) - y(n)]x(n)$$

  donde

  $$d(n) = \begin{cases} 1 & \text{si } x(n) \text{ pertenece a la clase } \zeta_1 \\ -1 & \text{si } x(n) \text{ pertenece a la clase } \zeta_1 \end{cases}$$

- *Paso 5:* Se incrementa n en una unidad y se vuelve al paso 2.

Como podemos ver, la adaptación del vector de pesos $w(n)$ es realizada en forma de regla de aprendizaje corrección de error, en la que η es el parámetro tasa de aprendizaje (parámetro que es positivo y que varía de 0 a 1) y la diferencia $d(n) - y(n)$ juega el papel de señal de error.

2.4. Redes multicapas

Una vez analizados los perceptrones se descubre que estos están muy limitados con respecto a lo que son capaces de hacer. Ante esta limitación es necesario aumentar la capacidad del perceptrón, lo cual solo es posible gracias a la adicción de capas adicionales. Estas capas son las denominadas anteriormente como capas ocultadas.

En las capas ocultas de una red multicapa se utilizan funciones no lineales ya que si se utilizaran todas lineales, estas se podrían agrupar en una lineal y sería similar a un perceptrón por lo que siempre incluirá alguna capa con una función no lineal.

En la práctica, las *feed-forward neural networks* (FFNNs) que son redes neuronales artificiales organizadas por capas sin realimentación son las más utilizadas y son denominadas en ocasiones por perceptrones multicapas.

Una red del tipo *feed-forward* tiene una estructura básica con al menos tres capas:

- Capa de entrada
- Capas ocultas
- Capa de salida

Normalmente todas las salidas de una capa conforman la entrada de la siguiente capa, sin embargo en redes como las convolucionales esto no funciona así ya que se crean conexiones locales que permiten que una capa reciba solo las salidas necesarias de la capa inmediatamente anterior.

Cuando existe más de una capa oculta se denomina la red en cuestión como profunda, es decir, deep neural network. Este el origen del término deep learning definido anteriormente.

Para entrenar una red multicapa el algoritmo que se suele usar es el algoritmo de propagación de errores conocido como *backpropagation* aunque realmente este término solo hace referencia a la propagación hacia atrás y no al algoritmo entero. Esta propagación hacia atrás lo que permite es calcular el gradiante del error con respecto a los parámetros de la red por lo que si se combina con una técnica de optimización como el gradiente descendiente se obtiene el algoritmo completo.

El gradiente se calcula para una función de error, llamada función de pérdida o *loss function* y el método de optimización lo que busca es minimizar este error.

A pesar de ser la más conocida no es la única técnica de aprendizaje de una red multicapa y estas técnicas se pueden agrupar en tres grandes bloques:

- Aprendizaje no supervisado: En este caso la red aprende por si sola diferentes patrones que encuentra en los datos mediante la detección de correlaciones. Nos ayuda a encontrar características propias de los datos de entrada.
- Aprendizaje por refuerzo: Esta técnica se denomina comúnmente como recompensa y castigos y es que la red aprende modificando los pesos individuales de la red para mejorar en el futuro y esto es posible gracias a que al final de la red tiene una recompensa o un castigo. Un ejemplo claro son los videojuegos en los que la red gana o pierde y a partir de ahí optimiza los parámetros.
- Aprendizaje supervisado: En esta ocasión la red recibe una realimentación directa para cada uno de los ejemplos del conjunto de entrenamiento. Esto permite que se puedan ajustar los parámetros de la red ya que sabe cuál es la salida deseada y para ellos se pueden usar técnicas de optimización usadas con *backpropagation*.

2.4.1. Propagación de los patrones de entrada

Las redes multicapas definen una relación entre las variables de entrada y las variables de salida de la red. Para obtener esta relación se propaga hacia adelante los valores de entrada. Cada neurona de la red procesa la información que le llega del exterior y produce una respuesta propagándola a las neuronas de la capa siguiente. Esta propagación de las respuestas se denominan también activaciones y a continuación se muestras las expresiones para calcularlas.

Sea una red multicapa con C capas (C-2 capas ocultas) y n_c neuronas en la capa c, para $c = 1, 2, ..., C$. Sea $W^c = (w^c_{ij})$ la matriz de pesos donde $W^c = (w^c_{ij})$ representa el peso de la conexion de la neurona i de la capa c para $c = 2, ..., C$. Denotaremos a^c_i a la activación de la neurona i de la capa c. Estas activaciones se calculan del siguiente modo:

- Activación de las neuronas de la capa de entrada (a^1_i). Las neuronas de la capa de entrada se encargan de transmitir hacia la red las señales recibidas desde el exterior. Por tanto:

  $$a^1_i = x_i \quad \text{para} \quad i = 1, 2, ...n_1$$

  donde $X = (x_1, x_2, ...x_{n1})$ representa el vector o patrón de entrada a la red.
- Activación de las neuronas de la capa oculta $c(a^c_i)$ Las neuronas ocultas de la red procesan la información recibida aplicando la función de activación f a la suma de

los productos de las activaciones que recibe por sus correspondientes pesos, es decir:

$$a_i^c = f\left(\sum_{j=1}^{n_{c-1}} w_{ji}^{c-1} a_j^{c-1} + u_i^c\right) \quad \text{para} \quad i = 1, 2, ..., n_c \text{ y } c = 2, 3, ...C-1$$

donde a_j^{c-1} son las activaciones de las neuronas de la capa $c-1$

- Activación de las neuronas de la capa de salida (a_i^C). Al igual que en el caso anterior, la activación de estas neuronas viene dada por la función de activación f aplicada a la suma de los productos de las entradas que recibe por sus correspondientes pesos:

$$y_i = a_i^C = f\left(\sum_{j=1}^{n_{C-1}} w_{ji}^{C-1} a_j^{C-1} + u_i^C\right) \quad \text{para} \quad i = 1, 2, ..., n_c$$

donde $Y = (y_1, y_2, ..., y_{n_C})$ es el vector de salida de la red.

La funcion f es la llamada *función de activación*. Para las redes multicapas, las funciones más utilizadas son la sigmoidal y la tangente hiperbólica, las cuales ya fueron expuestas anteriormente.

2.4.2. Algoritmo de Retropropagación

El algoritmo de aprendizaje es el mecanismo mediante el cual se ajustan los parámetros de la red. En este caso, el de una red multicapa, el algoritmo es de aprendizaje supervisado, es decir, ese ajuste se hará buscando la respuesta más próxima a la proporcionada en la entrada.

Puesto que el objetivo es que la salida de red sea lo más próxima a la salida deseada, el aprendizaje de la red se formula como un problema de minimización del siguiente modo:

$$Min_W E$$

siendo W el conjunto de parámetros de la red (pesos y umbrales) y E una función error que mide la diferencia entre las salidas de la red y las salidas deseadas. Normalmente, la función de error se define como:

$$E = \frac{1}{N}\sum_{n=1}^{N} e(n)$$

donde N es el número de muestras y $e(n)$ es el error cometido por la red para la muestra n, dado por:

$$e(n) = \frac{1}{n_C}\sum_{n=1}^{n_C}(s_i(n) - y_i(n))^2$$

siendo $Y(n) = (y_1(n), ..., y_{n_C}(n))$ y $S(n) = (s_1(n), ..., s_{n_C}(n))$ los vectores de salidas de la red y salidas deseadas para el patrón n, respectivamente.

De esta manera supongamos que $W*$ es un mínimo de la función de error E, en este punto el error es próximo a cero, lo cual implica que la salida de la red es próxima a la salida deseada, alcanzando así la meta de la regla de aprendizaje.

Por tanto, el aprendizaje de las redes multicapas es equivalente a encontrar un mínimo de la función de error. Las funciones de activación no lineales hace que las respuestas sean

no lineales con respecto a los parámetros que se ajustan por lo que es necesario el uso de técnicas de optimización no lineales para resolver el problema. Dichas técnicas en el contexto de redes neuronales y en particular para una red multicapa están basadas en el ajuste de los parámetros siguiendo una dirección, en este caso, en la dirección negativa del gradiente en la función E (método de descenso del gradiente), pues en el cálculo de varias variables, esta es la dirección en la que la función decrece.

Aunque el aprendizaje de la red se debe hacer para minimizar el error total, en la práctica se usan procedimientos basados en métodos del gradiente estocásticos que consisten en la minimización del error para cada patrón, $e(n)$. De este modo, si aplicamos el método de descenso del gradiente estocástico, cada uno de los parámetros w se ajusta para cada patrón de entrada n mediante la siguiente ley:

$$w(n) = w(n-1) - \alpha \frac{\partial e(n)}{\partial w}$$

donde α es la tasa de aprendizaje.

Lo que hace posible aplicar este método de forma eficiente es el hecho de que las neuronas de la red están agrupadas en capas de distintos niveles. Este hecho da lugar al llamado algoritmo de retropropagación o regla delta generalizada. El término retropropagación se utiliza debido a la forma de implementar el método del gradiente en la red multicapa, pues el error cometido en la salida de la red es propagado hacia atrás, transformándolo en un error para cada una de las neuronas ocultas de la red.

2.4.2.1. Regla delta generalizada

Para el desarrollo de la regla delta generalizada para el aprendizaje de una red multicapa es necesario distinguir dos casos ya que las reglas de modificación pueden variar.

Ajuste de pesos de la capa oculta $C-1$ a la capa de salida y umbrales de la capa de salida.

Sea w_{ji}^{C-1} el peso de la conexión de la neurona j de la capa $C-1$ a la neurona i de la capa de salida. Utilizando el método de descenso del gradiente, este parámetro se ajusta siguiendo la dirección negativa del gradiene del error:

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) - \alpha \frac{\partial e(n)}{\partial w_{ji}^{C-1}}$$

Por tanto, para la actualización de dicho parámetro se debe evaluar la derivada del error $e(n)$ en ese punto. Siguiendo la expresión del error y teniendo en cuenta, por un lado, que las salidas deseadas $s_i(n)$ para la red son constantes y no dependen del peso y, por otro lado, que el peso w_{ji}^{C-1} solo afecta a la neurona de salida i, $y_i(n)$, se obtiene que:

$$\frac{\partial e(n)}{\partial w_{ji}^{C-1}} = -(s_i(n) - y_i(n)) \frac{\partial y_i(n)}{\partial w_{ji}^{C-1}}$$

Como se observa se tiene que calcular la derivada de la neurona de salida $y_i(n)$ respecto al peso w_{ji}^{C-1}. La salida que se obtiene es la función de activación f aplicada a la suma de todas las entradas por sus pesos. Si aplicamos la regla de la cadena para derivar la composición de dos funciones y teniendo en cuenta que de todos los términos del sumatorio

el único que cuya derivada es distinta de cero es $w_{ji}^{C-1}a_j^{C-1}$, ya que interviene en el peso w_{ji}^{C-1}, se obtiene:

$$\frac{\partial y_i(n)}{\partial w_{ji}^{C-1}} = f'\left(\sum_{j=1}^{n_{C-1}} w_{ji}^{C-1}a_j^{C-1} + u_i^C\right) a_j^{C-1}(n)$$

Se define el término δ asociado a la neurona i de la capa de salida (capa C) y al patrón n, $\delta_i^C(n)$, del siguiente modo:

$$\delta_i^C(n) = -(s_i(n) - y_i(n))f'\left(\sum_{j=1}^{n_{C-1}} w_{ji}^{C-1}a_j^{C-1} + u_i^C\right)$$

Por tanto usando el valor de la derivada de la neurona de salida $y_i(n)$ y según el valor de $\delta_i^C(n)$ definido anteriormente, se obtiene que:

$$\frac{\partial e(n)}{\partial w_{ji}^{C-1}} = \delta_i^C(n)a_j^{C-1}(n)$$

Finalmente, reemplazando la derivada del error $e(n)$ respecto al peso w_{ji}^{C-1}, se obtiene la ley para modificar dicho peso.

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) + \alpha\delta_i^C(n)a_j^{C-1}$$

$$\text{para } j = 1, 2, ..., n_{C-1} i = 1, 2, ..., n_C$$

La Ley de Aprendizaje obtenida anteriormente para modificar los pesos de la última capa puede generalizarse para los umbrales de las neuronas de salida. Como se ha comentado, en las redes multicapas el umbral se trata como una conexión más cuya entrada es constante e igual a 1. Siguiendo entonces la ley anterior se puede deducir que los umbrales de las neuronas de la capa de salida se modifican con la siguiente expresión:

$$u_i^C(n) = u_i^C(n-1) + \alpha\delta_i^C(n) \quad \text{para} \quad i = 1, 2, ..., n_C$$

Adaptación de pesos de la capa c a la capa $c+1$ y umbrales de las neuronas de la capa $c+1$ para $c = 1, 2, ..., C-1$

Para facilitar el desarrollo de la regla de aprendizaje para el resto de pesos y umbrales, se elige un peso de la capa $C-2$ a la capa $C-1$. Sea w_{kj}^{C-1} el peso de la conexión de la neurona k de la capa $C-2$ a la neurona j de la capa $C-1$. Siguiendo el método de descenso del gradiente, la ley para actualizar dicho pesos viene dada por:

$$w_{kj}^{C-2}(n) = w_{kj}^{C-1}(n-1) + \alpha\frac{\partial e(n)}{\partial w_{kj}^{C-2}}$$

En este caso, y a diferencia del anterior, el peso w_{kj}^{C-2} influye en todas las salidas de la red, por lo que la derivada del error $e(n)$ respecto de dicho peso viene dada por la suma de las derivadas para cada una de las salidas de la red, es decir:

$$\frac{\partial e(n)}{\partial w_{kj}^{C-2}} = \sum_{i=1}^{n_C}(s_i(n) - y_i(n))\frac{\partial y_i(n)}{\partial w_{ji}^{C-1}}$$

Para calcular la derivada de la salida $y_i(n)$ respecto al peso w_{kj}^{C-2} es necesario tener en cuenta que este peso influye en la activación de la neurona j de capa oculta $C-1$, a_j^{C-1}, y que el resto de las activaciones de las neuronas en esta capa no depende de dicho peso. Por tanto, se tiene que:

$$\frac{\partial y_i(n)}{\partial w_{ji}^{C-2}} = f'\left(\sum_{j=1}^{n_{C-1}} w_{ji}^{C-1} a_j^{C-1} + u_i^C\right) w_{ji}^{C-1} \frac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}}$$

Teniendo en cuenta este valor y de acuerdo con la definición de δ, obtenemos:

$$\frac{\partial e(n)}{\partial w_{kj}^{C-2}} = \sum_{i=1}^{n_C} \delta_i^C(n) w_{ji}^{C-1} \frac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}}$$

Para obtener la ley de aprendizaje para el peso w_{kj}^{C-2} solo falta derivar la activación de la neurona j de la capa oculta $C-1$,a_j^{C-1} , respecto a dicho peso. De nuevo, aplicando la regla de la cadena:

$$\frac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}} = f'\left(\sum_{k=1}^{n_{C-2}} w_{kj}^{C-2} a_k^{C-2} + u_j^{C-2}\right) a_k^{C-2}(n)$$

Se define el valor δ para neuronas de la capa $C-1$, $\delta^{C-1}(n)$ como:

$$\delta_j^{C-1}(n) = f'\left(\sum_{k=1}^{n_{C-2}} w_{kj}^{C-2} a_k^{C-2} + u_j^{C-2}\right) \sum_{i=1}^{n_C} \delta_i^C(n) w_{ji}^{C-1}$$

Usando la definición de este valor tenemos que:

$$\frac{\partial e(n)}{\partial w_{kj}^{C-2}} = \delta_j^{C-1}(n) a_k^{C-2}(n)$$

Y como consecuencia, la ley de aprendizaje para modificar el peso w_{kj}^{C-2} viene dada por:

$$w_{kj}^{C-2}(n) = w_{kj}^{C-2}(n-1) + \alpha \delta_j^{C-1}(n)_k^{C-2}(n)$$

$$\text{para } k = 1, 2, ..., n_{C-2} \text{ y } j = 1, 2, ..., n_{C-1}$$

Del mismo modo que la ley par modificar los pesos de la última capa se observa que para modificar el peso de la conexión de la neurona k de la capa $C-2$ a la neurona j de la capa $C-1$ solo es necesario considerar la activación de la neurona de la que parte la conexión y el término δ de la neurona a la que llega la conexión. La diferencia está en la expresión de δ. En este punto, es posible generalizar la ley para los pesos de la capa c a la capa $c+1$ ($c = 1, 2, ..., C-2$).

$$w_{kj}^c(n) = w_{kj}^c(n-1) + \alpha \delta_j^{c+1}(n) a_k^c(n)$$

$$\text{para } k = 1, 2, ..., n_c, j = 1, 2, ..., n_{c+1} \text{ y } c = 1, 2, ...C-2$$

donde $a_k^c(n)$ es la activación de la neurona k de la capa c para el patrón n y $\delta_j^{c+1}(n)$ viene dado por la siguiente expresión:

$$\delta_j^{c+1}(n) = \left(\sum_{k=1}^{n_c} w_{kj}^c a_k^c + u_j^c\right) \sum_{i=1}^{n_{c+1}} \delta_i^{c+2}(n) w_{ji}^c$$

Es posible, también, generalizar la ley de aprendizaje para el resto de umbrales de la red; basta tratarlos como conexiones cuya entrada es constante e igual a 1. La ley para modificarlos viene dada por:

$$u_j^{c+1}(n) = u_j^{c+1}(n-1) + \alpha\delta_j^{c+1}(n)$$

$$\text{para } j = 1, 2, ..., n_{c+1} \text{ y } c = 1, 2, ...C-2$$

Derivada de la función de activación

Para calcular los valores de δ para cada neurona de la red multicapa es necesario derivar la función de activación por lo que este valor dependerá de la misma. Como ya se ha comentado en redes multicapas las más usadas son la sigmoidal y la tangente hiperbólica. A continuación se van a calcular las derivadas para estas dos funciones para poder completar el calculo de los valores de δ.

- Derivada de la función sigmoidal.

  Derivando la expresión de la función sigmoidal, se obtiene que :

  $$f_1'(x) = \frac{1}{(1+e^{-x})^2}(-e^{-x}) = \frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}}$$

  Por tanto,

  $$f_1'(x) = f_1(x)(1-f_1(x))$$

  Como consecuencia, cuando se utiliza la función sigmoidal, los valores δ asociados a las neuronas de salida tienen la siguiente forma:

  $$\delta_i^C(n) = -(s_i(n) - y_i(n))y_i(n)(1-y_i(n)) \quad \text{para} \quad i = 1, 2, ..., n_C$$

  Y los valores de δ para el resto de las neuronas de la red , vienen dados por:

  $$\delta_j^{c+1}(n) = a_j^c(n)(1-a_j^c(n)) \sum_{i=1}^{n_{c+1}} \delta^{c+2}(n) w_{ji}^c$$

  $$\text{para } j = 1, 2, ..., n_{c+1} \text{ y } c = 1, 2, ...C-2$$

- Derivada de la función tangente hiperbólica

  Teniendo en cuenta que $f_2(x) = 2f_1(x) - 1$, la derivada de la función $f_2(x)$ es:

  $$f_2'(x) = 2f_1(x)(1-f_1(x))$$

  Cuando se utilice, por tanto, la función de activación tangente hiperbólica, los valores de δ para las neuronas de la red adoptan las expresiones dadas por las ecuaciones calculadas para la función sigmoidal pero multiplicadas por un factor de 2.

A pesar de que las funciones de activación más usadas son estas, normalmente en la neurona de salida se suele utilizar la función identidad, es decir,$f(x) = x$. En este caso, la derivada de la función de activación en la salida es 1 y, como consecuencia, los valores de δ para las neuronas de salida adoptan la siguiente expresión:

$$\delta_i^C(n) = -(s_i(n) - y_i(n)) \quad \text{para} \quad i = 1, 2, ..., n_C$$

2.4.2.2. Proceso de aprendizaje de una red multicapa

Como ya se ha comentado el objetivo del entrenamiento de una red multicapa es ajustar los parámetros de la red con el fin de que las entradas presentadas produzcan las salidas deseadas, es decir, con el fin de minimizar la función de error E. Se va a detallar el proceso completo de aprendizaje de acuerdo al algoritmo de retropropagación.

Sea $(X(n), S(n)), n = 1, ...N$ el conjunto de muestras o patrones que representan el problema a resolver, donde $X(n) = (x_1(n), ..., x_{n1}(n))$ son los patrones de entrada a la red, $S(n) = (s_1(n), ..., s_{nc}(n))$ son las salidas deseadas para dichas entradas y N es el número de patrones disponibles. Generalmente, es frecuente encontrar los patrones de entrada y salida normalizados o escalados mediante una transformación lineal en los intervalos $[0, 1]$ o $[-1, 1]$ dependiendo de la función de activación empleada.

Los pasos que componen el proceso de aprendizaje del perceptrón multicapa son los siguientes:

1. Se inicializan los pesos y umbrales de la red. Esta inicialización es aleatoria y con valores alrededor de cero.
2. Se toma un patrón n del conjunto de entrenamiento, $(X(n), S(n))$, y se propaga hacia la salida de la red el vector de entrada $X(n)$, obteniéndose así la respuesta de la red para dicho vector de entrada, $Y(n)$.
3. Se evalúa el error cuadrático cometido por la red para el patrón n.
4. Se aplica la regla delta generalizada para modificar los pesos y umbrales de la red.

   Para ello se siguen los siguientes pasos:

   a) Se calculan los valores δ para todas las neuronas de la capa de salida.

   b) Se calculan los valores δ para el resto de las neuronas de la red empezando desde la última capa oculta y retropropagando dichos valores hacia la capa de entrada.

   c) Se modifican los pesos y umbrales de la red para los pesos y umbrales de la capa de salida y para el resto de parámetros de la red.
5. Se repiten los pasos 2, 3 y 4 para todos los patrones de entrenamiento, completando así un ciclo de aprendizaje.
6. Se evalúa el error total E cometido por la red. Dicho error también recibe el nombre de error de entrenamiento, pues se calcula utilizando los patrones de entrenamiento.
7. Se repiten los pasos 2, 3, 4, 5 y 6 hasta alcanzar un mínimo del error de entrenamiento, para lo cual se realizan m ciclos de aprendizaje

2.4.2.3. Early stopping

Una manera de acometer el problema de sobre-entrenamiento (*overfitting*) es extraer un subconjunto de muestras del conjunto de entrenamiento y utilizarlo de manera auxiliar durante el entrenamiento. Este subconjunto recibe el nombre de conjunto de validación. La función que desempeña el conjunto de validación es evaluar el error de la red tras cada ciclo (o tras cada cierto número de ciclos) y determinar el momento en que este empieza a aumentar.

Ya que el conjunto de validación se deja al margen durante el entrenamiento, el error cometido sobre él es un buen indicativo del error que la red cometerá sobre el conjunto de test. En consecuencia, se procederá a detener el entrenamiento en el momento en que el error de validación aumente y se conservarán los valores de los pesos del ciclo anterior. Este criterio de parada se denomina *early-stopping*. La siguiente figura describe el procedimiento explicado.

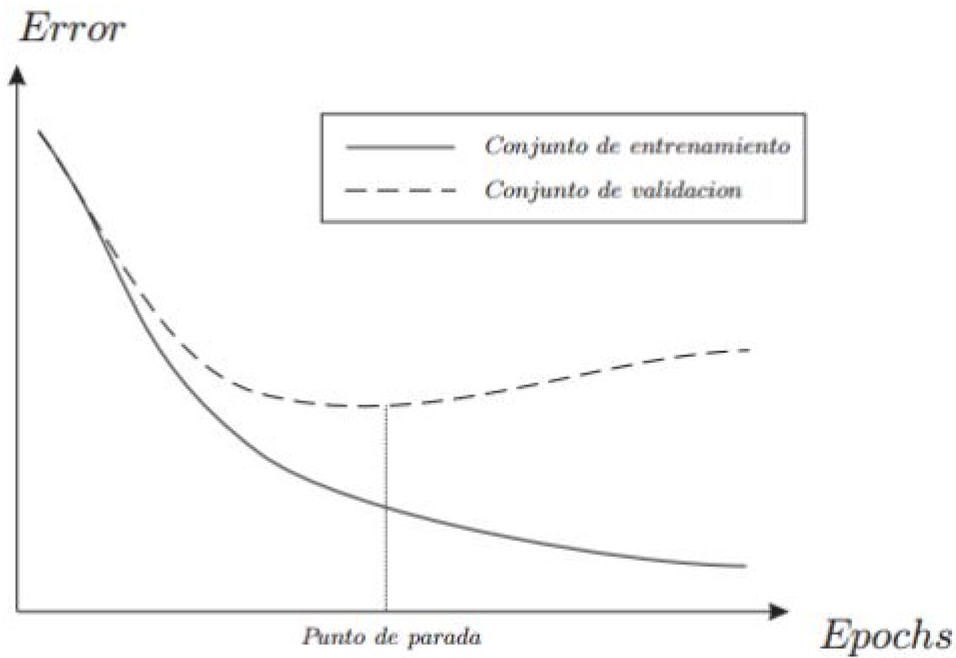


Figura 2.7: Representación del criterio de early-stopping

Capítulo 3

Regularización del Deep Learning

A pesar de que teóricamente una red neuronal multicapa puede ser un predictor que funciona de manera universal capaz de aprender cualquier cosa que se le plantee, la realidad es que en la práctica no puedes llegar a tener la certeza de cuáles son los hiperparámetros óptimos a usar en el entrenamiento para conseguir un aprendizaje adecuado. Nos encontramos entonces con las dificultades para generalizar el algoritmo y que funcione fuera del conjunto de datos de entrenamiento.

Decimos que estamos ante una situación de sobreaprendizaje cuando conseguimos un error pequeño sobre los datos de entrenamiento pero cuando usamos ese modelo sobre datos que no se han usado antes (conjunto de datos de test) el error aumenta bastante provocando así fallos en las predicciones. Para evitar esto y conseguir bajar el error de generalización de nuestro modelo se usan técnicas de regularización.

3.1. El error de generalización

El error de generalización de un modelo de aprendizaje automático es la diferencia entre el error empírico y el error esperado. En la práctica, se mide por la diferencia entre el error de los datos de entrenamiento y el de los datos de prueba. Esta medida representa la capacidad del modelo para generalizar a partir de los datos de entrenamiento a nuevos datos antes no vistos. El primer paso para conseguir esto es que el modelo no memorice los datos de entrenamiento sino que aprenda reglas a partir de estos, siendo capaz de extrapolarla después. El error de generalización de las redes neuronales profundas (DNN) son actualmente el foco de una investigación centrada en reducir dicho error. Estas redes han demostrado un enorme rendimiento en diversos campos como la visión artificial o el reconocimiento de voz. A pesar de no estar garantizado el buen rendimiento suelen lograr pequeños errores en el set de entrenamiento y en la generalización.

Consideramos un problema de clasificación donde el espacio de entrada de un algoritmo de aprendizaje es el subespacio $\mathcal{D}$-dimensional $\mathcal{X} \subseteq \mathbb{R}^{\mathcal{D}}$ y $x \in \mathcal{X}$ es una muestra de datos de entrada para el algoritmo. El espacio de salida es el subespacio K-dimensional $\mathcal{Y} \subseteq \mathbb{R}^{K}$. La etiqueta de la muestra de entrada x es $y \in \mathcal{Y}$. El conjunto de muestras se denota como $\mathcal{Z}$, donde $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$.

Al encontrarnos en un problema de clasificación, K determina el número de clases posibles y $k^* \in \{1, ..., K\}$ es la clase correcta de la muestra de entrada x. En consecuencia,

el vector de etiqueta $y \in \mathcal{Y}$ es un vector one-hot , lo que significa que todos sus elementos son iguales a cero excepto por el elemento del $k^* - ésimo$ índice que es igual a uno.

Supongamos ahora que un conjunto de datos de entrenamiento tiene N ejemplos de entrenamiento, de manera que el conjunto $S_N = \{s_i | s_i \in \mathcal{Z}\}_{i=1}^{N} = \{(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}\}_{i=1}^{N}$ es el conjunto de entrenamiento del algoritmo. Las muestras se extraen de forma independiente de una distribución de probabilidad $\mathcal{D}$. El conjunto $t_{N_{test}} = \{t_i | t_i \in \mathcal{Z}\}_{i=1}^{N_{test}} = \{(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}\}_{i=1}^{N_{test}}$ es el conjunto de prueba del algoritmo, que consta de N_{test} ejemplos de prueba. El conjunto de hipótesis, que consta de todas las funciones posibles f_W , se denota como $\mathcal{H}$. Por lo tanto, un algoritmo de aprendizaje $\mathcal{A}$ es un mapeo de $\mathcal{Z}^N$ a $\mathcal{H}$, es decir, $\mathcal{A} : \mathcal{Z}^N \to \mathcal{H}$.

La función de pérdida, que mide la discrepancia entre la etiqueta verdadera y la etiqueta estimada del algoritmo $f(x)$ se denota por $\ell(y, f(x))$. En algunos casos, cuando se refiere a la pérdida de un algoritmo de aprendizaje específico $\mathcal{A}$ entrenando en el conjunto S_N y evaluado en la muestra s denotamos en su lugar $\ell(\mathcal{A}_{S_N}, s)$

Para una función de pérdida general, ℓ, la pérdida empírica de un algoritmo en el conjunto de entrenamiento es:

$$\ell_{emp}(f, S_N) = \ell_{emp}(\mathcal{A}_{S_N}) = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, f(x_i)), \quad (x_i, y_i)_{i=1}^{N} \in S_N$$

y la pérdida esperada del algoritmo es:

$$\ell_{emp}(f) = \ell_{emp}(\mathcal{A}_{S_N}) = \mathbb{E}_{(x,y) \sim \mathcal{D}}[\ell(y, f(x))]$$

En consecuencia, el error de generalización de un algoritmo viene dado por:

$$GE(f, S_N) = |\ell_{emp}(f, S_N) - \ell_{emp}(f, S_N).$$

3.2. El problema del sobreaprendizaje

En general el sobreaprendizaje aparece cuando tenemos demasiados atributos de los datos que hace difícil poder extraer una regla generalizable debido a que muchos de los atributos aportan detalles que no son realmente relevantes para el modelo deseado. Sin embargo, este problema también puede aparecer en los modelos más simples que queramos entrenar. Esto se debe principalmente a que el número de hipótesis crece exponencialmente con el número de atributos o características que tengamos. Supongamos un conjunto de datos en los que se tienen n características binarias, estas dan lugar a 2^n patrones distintos. Formalmente podemos decir que el sobreaprendizaje

En el conjunto de entrenamiento además de los patrones útiles podemos encontrar ruidos, es decir, regularidades debidas al conjunto de datos utilizado (error de muestreo). Al ajustar los hiperparámetro de una red neuronal no podemos diferenciar cuales de estas regularidades son reales y cuales debidas a errores de muestreo por lo que siempre corremos el riesgo de tener sobreaprendizaje. Como se ha dicho anteriormente, una señal que no deja posible duda de que existe sobreaprendizaje es cuando el error sobre los datos de entrenamiento es mucho menor que sobre los datos de prueba.

Realmente este problema se podría reducir al problema de sesgo y varianza. El sobreaprendizaje aparece cuando la varianza es elevada. La clave está en llegar a un equilibrio

entre el sesgo y la varianza que nos permita obtener el rendimiento óptimo. Como ya se ha comentado, las redes neuronales tienen muchos parámetros y como consecuencia los modelos pueden tener una elevada varianza.

Para evitar el sobreaprendizaje podemos utilizar tres estrategias, también conocidas como técnicas para reducir la varianza:

- *Utilizar más datos* En *deep learning* suele ser la mejor opción, siempre que se tenga la capacidad de cálculo necesaria.
- *Ajustar la capacidad del modelo* En el entrenamiento de un modelos existen muchos detalles que son importantes para obtener el rendimiento deseado, como son las funciones de activación, la inicialización de sus pesos, etc. Por este motivo se debe ser muy cuidadoso a la hora de establecer los hiperparámetros. Un buen ajuste de esos parámetros sirve para que la red obtenga la capacidad adecuada para identificar las regularidades relevantes sin que esto implique ajustarse a los nuevos datos y tener sobreajuste.
- *Combinar múltiples modelos* Una última estrategia para evitar este problema es combinar múltiples modelos para que así los errores de cada uno se compensen con los aciertos de los demás En las redes neuronales se puede realizar de dos formas:
    - Promediando múltiples modelos [*model averaging*]:Se construyen muchos modelos diferentes con diferentes parámetros o el mismo tipo de modelo utilizando distintos subconjuntos del conjunto de entrenamiento (p.ej. bagging). Es decir, se crea un ensemble.
    - Aplicando un enfoque bayesiano [*Bayesian fitting*], se selecciona una única arquitectura de red. Después se van combinando las predicciones resultantes de entrenar la red en múltiples ocasiones. Es decir, en lugar de entrenar la red con distintos subconjuntos de entrenamiento, entrenamos múltiples versiones de una misma red neuronal.

A pesar de ser un problema común, existen ciertas situaciones en las que no nos debemos preocupar por él, pero sí por otros problemas. Cuando nuestro conjunto de datos crece más rápido que nuestra capacidad para procesarlos, usamos cada ejemplo de entrenamiento una sola vez por lo que nuestros problemas podrían ser de sesgo o de falta de capacidad (underfitting).

3.3. Regularización

Se denomina regularización a cualquier cambio que se realice en un algoritmo de aprendizaje con la finalidad de reducir su error de generalización pero no su error de resustitución (error sobre el conjunto de entrenamiento). Es decir, podemos decir que la regularización consiste en prevenir el sobreaprendizaje.

Podemos utilizar dos técnicas de regularización para obtener este objetivo.

- Añadir restricciones al modelo. En esta técnica se incluyen restricciones en sus hiperparámetros. Por ejemplo, hacer que se compartan parámetros entre diferentes capas de la red.
- Añadir términos adiciones a la función de coste. Se puede interpretar como una manera de restringir más débil que la anterior.

No centraremos en la segunda estrategia.

3.3.1. Regularización de la funcion de coste

La manera más utilizada para regularizar el entrenamiento de una red neuronal artificial es la adición de términos extras a la función de coste elegida. Esta función era una función de error con la que intentábamos que la red aprendiese evaluando siempre el conjunto de entrenamiento. Cuando se usa un término de regularización la función de coste es la siguiente:

$$L = L_E + \lambda L_R$$

donde $L_E = E$ es nuestra función de error, L_R es el término de regularización y λ es un parámetro de regularización que nos sirve para darle más o menos peso al término de regularización con respecto al término L_E asociado al error

El modo de entrenar la red no varía, se calcula el gradiente de la función de coste:

$$\nabla_x L = \nabla_x L_E + \lambda \nabla_x L_R$$

El proceso restante sigue siendo el mismo excepto la señal propagada hacia atrás. En esta señal se incluirá información que contribuya a bajar el error de generalización. Observamos que al utilizar esta técnica un nuevo hiperparámetros se ha incluido en el algoritmo. El hiperparámetro λ hará posible controlar la capacidad de la red.

Capítulo 4

Redes Neuronales Convolucionales (CNN)

4.1. Introducción

Las redes neuronales convoluciones o CNNs (Convolutional Neural Network) son un tipo particular de redes neuronales artificiales cuya especialización son el procesamiento de datos referidos a una señal. Podemos considerar señal todo mensaje o información que se transmita a través de imágenes. Este tipo de red es una variación de un perceptrón multicapa donde imitan el cortex visual del ojo humano para realizar tareas de visión artificial.

El origen de las CNNs lo encontramos en el Neocognitron de Kunihiko Fukushima en 1980, cuya arquitectura es muy similar a las redes convolucionales actuales. La diferencia existente es que los modelos de Fukushima no incluian un algoritmo de entrenamiento óptimo.Posteriormente, Yann LeCunn mejoró este primer modelo introduciendo el aprendizaje basado en la propagación hacia atras, bockpropagation. Sin embargo, debido a las limitaciones computacionales de los años 90 el número de redes era pequeño. Más tarde, en 2012 llegó Dan Ciresan, entre otros, para refinar este tipo de redes implementándolas en GPU obteniendo así un mejor rendimiento.

En la historia del Deep Learning es claro que estas redes han jugado un papel muy importante ya que se puede considerar la aplicación con mayor éxito en el ámbito de obtener algoritmos imitando el cerebro humano. Uno de los primeros ejemplos que se pueden encontrar es el desarrollo de una red convolucional capaz de leer cheques. Esta aplicación fue desarrollada por un grupo de investigadores de AT&T. Una de las principales barreras que se encontraron los desarrollados de este tipo de redes fueron psicológicas ya que aparentemente era imposible imitar el cerebro humano para este tipo de modelos. Sin embargo, a día de hoy se han convertido en las redes por excelencia que hicieron el camino más fácil a las redes neuronales en general.

Para ver las diferencias existentes entre estas redes y el resto de redes hay que fijarse en sus entradas y sus salidas. Estas suelen ser estructuradas, hecho que aprovecharán las redes convolucionales para tener arquitecturas más eficientes. Estas redes constan de diversas capas convolucionales y de reducción (*pooling*) que aparecen de manera alternadas. Finalmente, tiene una serie de capas full-coneceted como una red perceptrón multicapa.

La entrada a una red convolucional suele ser una imagen m x m x r, donde m es la altura y ancho de la imagen y r son los canales. La salida puede ser también una imagen, como cuando queremos detectar zonas concretas en unas imágenes, o pueden ser convencionales cuando se trate de un problema de clasificación o regresión.

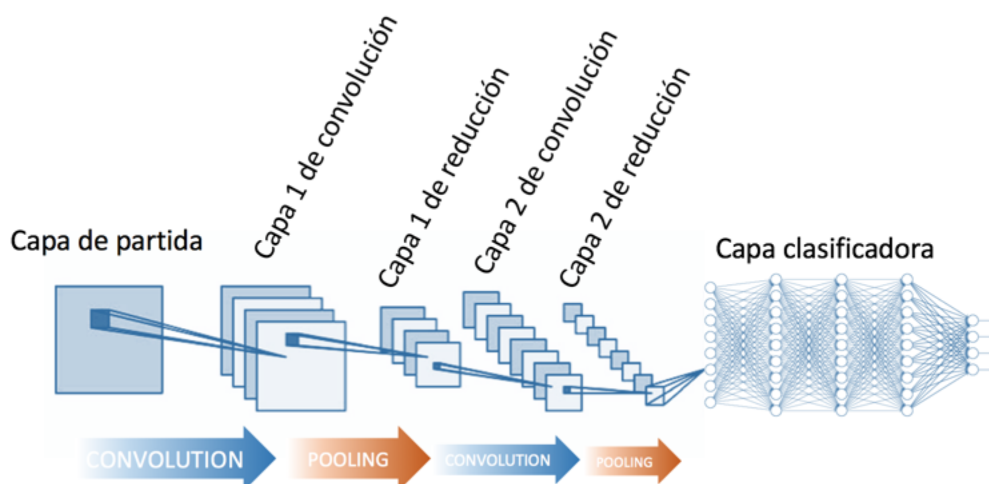


Figura 4.1: Estructura clásica de una red convolucional para clasificación

4.2. La operación de convolución

El nombre de redes neuronales convoluciones tiene su origen en la operación que se realiza en algunas de sus capas. Se sustituye la operación de multiplicación matricial de entradas por pesos por la operación matemática de convolución.

Si hablamos matemáticamente, una convolución es una operación aplicada a dos funciones con números reales como argumentos. La salida de esta operación es una tercera función que se interpreta como la modificada de una de las funciones de entrada.

Sea f y g dos funciones, se denota $f * g$ y se define como la integral del producto de ambas funciones después de desplazar una de ellas una distancia t. El intervalo de integración depende del dominio de las funciones.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$

Al trabajar con imágenes en un ordenador, los datos de los que dispondremos serán discretos por lo que la integral anterior deberá convertirse en un sumatorio de funciones contínuas, de la siguiente forma:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]$$

En aplicaciones de Deep Learning, el primer término de la convolución es la imagen que queremos procesar, $x[n]$, mientras que el segundo es el kernel con el que se procesa,$h[n]$. Si el dominio del kernel es finito, se define la convolución sobre el dominio $0, 1, ..., K-1$

y la convolución consiste en para cada valor de la imagen realizar K multiplicaciones y $K-1$ sumas:

$$(x*h)[n] = \sum_{k=0}^{K-1} h[k]x[n-k]$$

Para el tratamiento de imágenes se debe generalizar para un caso multidimensional. Las variables $[n1, n2]$ corresponderían a las coordenadas de los píxeles de la imagen. Los kernels utilizados tienen tamaños K_1 y K_2

$$(x*h)[n_1, n_2] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} h[k_1, k_2]x[n_1+k_1, n_2+k_2]$$

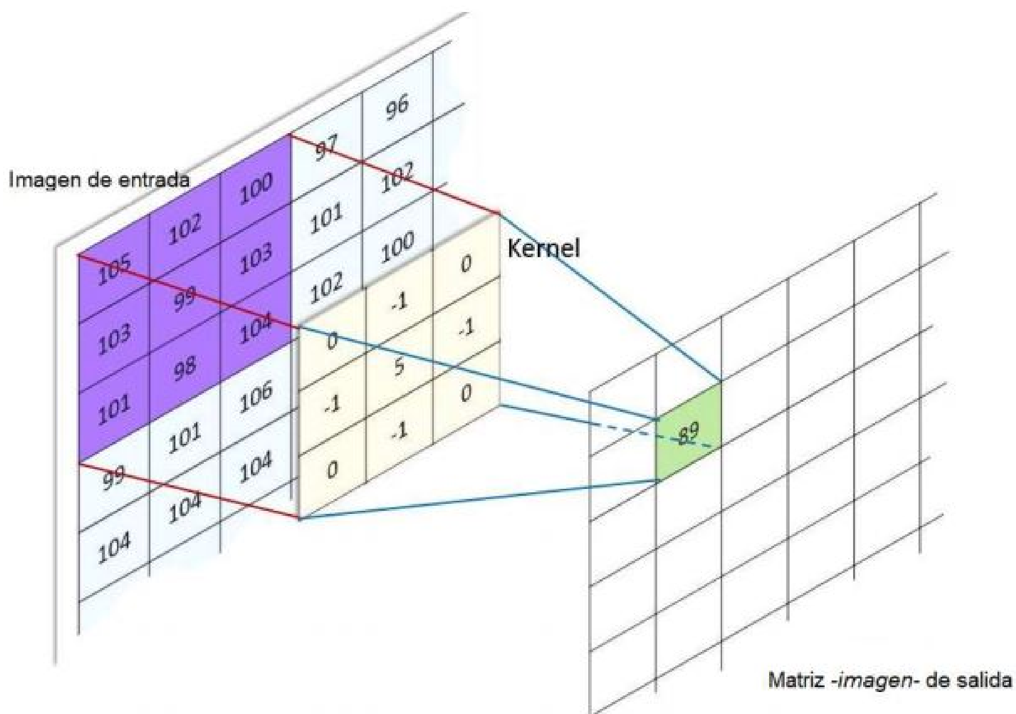


Figura 4.2: Ejemplo de operación de convolución

Ejemplifiquemos una operación de convolución con una matriz 7 x 7 de los píxeles de una imagen en blanco y negro.

$$I = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Definimos a continuación el kernel que se le va a aplicar:

$$H = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Para realizar la convolución debemos situar el kernel en diferentes posiciones de la imagen anterior y de esta manera obtendremos la imagen filtrada. Por ejemplo, utilicemos el kernel en la siguiente posición:

$$\begin{bmatrix} - & - & - & - & - & - & - \\ - & 0 & 0 & 0 & - & - & - \\ - & 0 & 1 & 1 & - & - & - \\ - & 0 & 1 & 1 & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \\ - & - & - & - & - & - & - \end{bmatrix}$$

A continuación multiplicamos elemento a elemento y sumamos el resultado:

$$0*0-1*0+0*0-1*0+4*1-1*4+0*0-1*1+0*1=4-1-1=2$$

El resto de operaciones donde se pueda colocar el kernel son análogas, obteniendo como resultado la siguiente matriz:

$$I*H = \begin{bmatrix} 0 & -1 & -1 & -1 & 0 \\ -1 & 2 & 1 & 2 & -1 \\ -1 & 1 & 0 & 1 & -1 \\ -1 & 2 & 1 & 2 & -1 \\ 0 & -1 & -1 & -1 & 0 \end{bmatrix}$$

Si nos fijamos la matriz resultante es más pequeña que la original. En caso de querer obtener una imagen de las mismas dimensiones podemos rodear la imagen por ceros, tantos como nos hagan falta:

$$I *_{zp} H = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & -1 & 0 & 0 \\ 0 & -1 & 2 & 1 & 2 & -1 & 0 \\ 0 & -1 & 1 & 0 & 1 & -1 & 0 \\ 0 & -1 & 2 & 1 & 2 & -1 & 0 \\ 0 & 0 & -1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

4.3. Capas convolucionales

Como ya se ha comentado anteriormente, lo que diferencia una red convolucional de cualquier otra es que sustituye la multiplicación de matrices aplicada por la operación de convolución. Es por eso que recibe el nombre de capa convolucional.

La entrada de estas capas son las imágenes, las cuales se procesan con el kernel correspondiente llevando a cabo la convolución. Otras de las diferencias que presentan estas capas es que no todas las imágenes de entrada están conectadas con las imágenes de salida, sino que cada neurona de una capa realiza la operación sobre un conjunto reducido de imágenes de entrada.

Para optimizar y mejorar este procedimiento se hace uso principalmente de tres ideas:

- Interacciones dispersas: Como el kernel aplicad tiene un tamaño menor que la imagen de entrada, esto hace que se reduzca el número de parámetros.
- Parámetros replicados: Se utilizan los mismos parámetros, pesos por ejemplos, en diferentes neuronas donde las entradas están conectadas a otras regiones de la imagen.
- Actividad neuronal equivariante, es decir, implica que si las entradas cambian, las salidas van a cambiar de manera similar.

Matemáticamente, una función es equivariante si, cuando la entrada cambia, la salida cambia de la misma forma:

$$f(g(x)) = g(f(x))$$

Una ventaja de este tipo de capas es la capacidad de trabajar con imágenes de diferentes tamaño como entrada.

4.4. Capas de Pooling

Un aspecto importante es el elevado coste computacional que conlleva procesar imágenes de una red neuronal. Para solventar este problema, se utilizan las llamadas capas de Pooling o reducción de muestreo. La funcián de estas capas es reducir las dimensiones espaciales(ancho y alto) sin afectar a la profundidad para que las capas posteriores reciban datos de menor tamaño. Esta reducción produce que se pierda información, aunque tiene ventajas:

- Disminución del tamaño para tener una menor sobrecarga de cálculo en las capas posteriores.
- Reducción del sobreajuste.

Para realizar esta reducción de muestreo existen dos tipos de funciones destacadas que se suelen usar:

- Técnica *Max - pooling*: Esta operación divide la imagen de entrada en un conjunto de rectángulos y va escogiendo el valor máximo. La forma más común de realizarlo es aplicarlo en zonas 2 x 2 lo que reduce la muestra un 75
- Técnica *Average - pooling*: En esta operación lo que se hace es coger la media aritmética de los valores de la región en lugar del máximo.

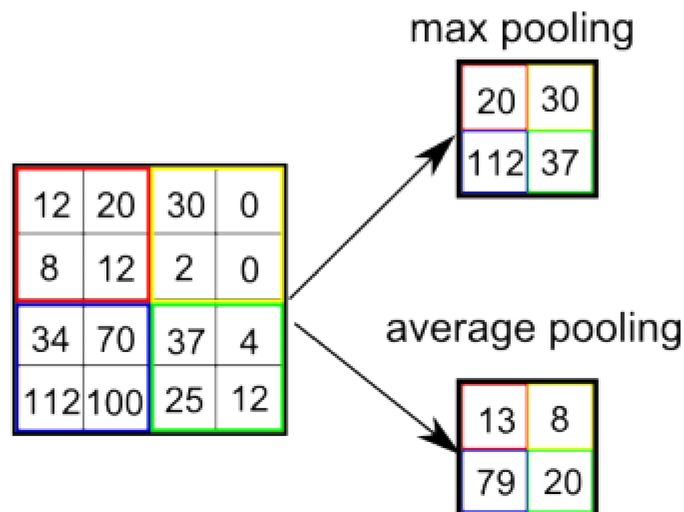


Figura 4.3: Ejemplo de reducción de muestreo

4.5. Capas full-conected

Tras haber pasado la imagen de entrada por las capas de convolución y reducción, normalmente se añaden 2 o más capas denominadas Fully connected o totalmente conectada. Deben este nombre a que todas las neuronas en una capa totalmente conectada se conectan a las neuronas de la capa anterior.

En problemas de clasificación, la última capa Fully connected se encarga de combinar todos los parámetros aprendidos a través de las capas anteriores para clasificar las imágenes. Por este motivo, el parámetro conocido como *Ooutput-Size* (Tamaño de salida) es el número de clases que se desea predecir. Por otro lado, en problemas de regresión el tamaño de salida es igual al número de variables de respuesta que exista.

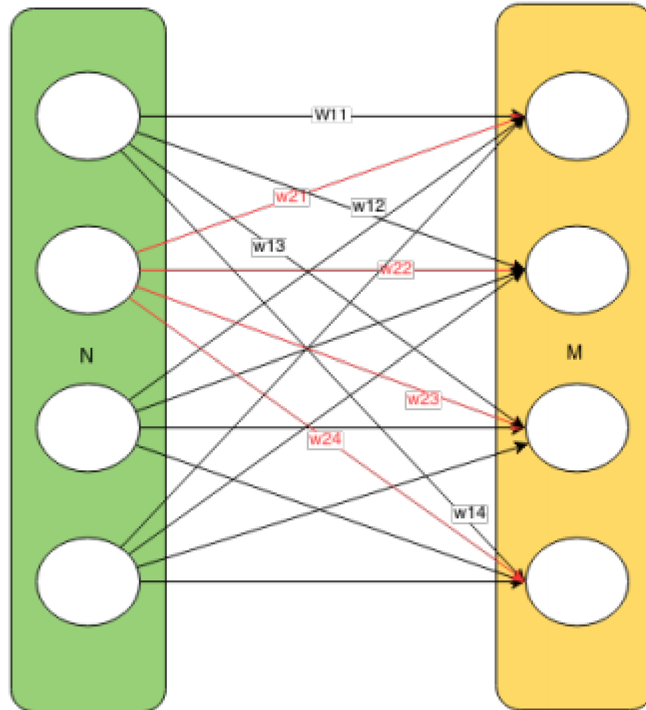


Figura 4.4: Ejemplo de capa totalmente conectada

4.6. Capas de softmax y de clasificacion

Si nos centramos en problemas de clasificación, tras la última capa Fully Conected son necesarias una capa *softmax* y una capa de clasificación. La función de activación de la unidad de salida para esta capa es la función *softmax*.

Por último, la capa de clasificación toma los valores de la función *softmax* y asigna cada entrada a una de la clases existentes mutualemte excluyentes usando la probabilidad de pertenencia a las clases.

Capítulo 5

La visión artifical y las CNNs

La visión por computadora ha sido siempre una área de investigación muy activa en las aplicaciones del Machine Learning ya que aunque para los humanos y animales resulta una tarea sencilla, es un verdadero desafío para los ordenadores.

Gracias a todos los avances que se han conseguido en la visión artificial ha llegado a convertirse en una herramienta usada en muchos campos de trabajo en la actualidad. Entre los campos que más la utilizan destacaría la medicina, la vigilancia y seguridad, la industria, la automoción y la robótica.

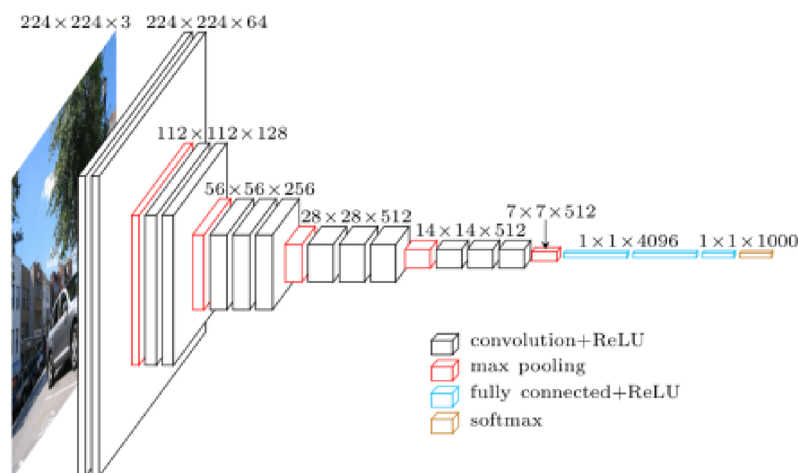


Figura 5.1: Macroarquitectura de la red VGG16

5.1. Preprocesamiento

La mayoría de las aplicaciones en las redes neuronales convolucionales requiere un preprocesamiento ya que la entrada que se usa suele venir en una forma difícil de representar para la mayoría de las arquitecturas del Deep Learning. La visión por computadora generalmente requiere poco preprocesado.

Las imágenes, datos de entrada en estas aplicaciones, se deben estandarizar para que los píxeles estén en el mismo rango, por ejemplo [0,1] o [-1,1]. Este preprocesado es en realidad el único necesario. A pesar de que la mayoría de arquitecturas requieres que todas

sus imágenes tengan el mismo tamaño, algunos modelos convolucionales son ya capaces de aceptar entradas de tamaño variable y de manera dinámica ajustan el tamaño de las entradas para que se mantenga constante el tamaño de salida.

Un aumento del conjunto de datos de entrada es una excelente forma de reducir el error de generalización en la mayoría de los modelos. Una práctica común en esto casos es introducir en el modelo diferentes versiones de la misma entrada(por ejemplo recortada en posiciones diferentes).

5.2. Normalización del contraste

Una de las fuentes de variación en el conjunto de entradas y que conviene eliminar en la mayoría de las tares es la cantidad de contraste de la imagen. El contraste no es más que la diferencia entre los píxeles brillantes y los oscuros de una imagen. Existen diferentes formar de cuantificar esta magnitud. Si hablamos en el contexto del Deep Learning suele estar cuantificado como la desviación estándar de los píxeles en una región concreta de la imagen.

Supongamos que tenemos una imagen representada por un tensor $\mathtt{X} \in \mathbb{R}^{rxcx3}$, siendo $X_{i,j,1}$ la intensidad roja la fila i y la columna j, siendo $X_{i,j,2}$ la intensidad verde y dando $X_{i,j,3}$ la intensidad azul.

Entonces el contraste de toda la imagen viene dado por:

$$\sqrt{\frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(X_{i,j,k}-\bar{\mathtt{X}})^2}$$

donde $\mathtt{X}$ es la intensidad media de toda la imagen:

$$\mathtt{X} = \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}X_{i,j,k}$$

El objetivo de la denominada normalización de contraste global (GCN) es conseguir que las imágenes no varíen de contraste. Para ellos se resta la media de cada imagen y luego se vuelve a escalar para que la desviación estándar en sus píxeles sea igual a una cierta constante s.

Uno de los inconvenientes que se pueden encontrar es la imposibilidad para cambiar el contraste de una imagen de contraste cero (una cuyo píxeles tienen la misma intensidad). Además una imagen con contraste muy bajo en general tienen muy poco contenido y al realizar la división entre la desviación estándar lo normal es que lo que se consiga sea aumentar el ruido.

Para solucionar este problema se introduce un parámetro de regulación positivo λ para sesgar la estimación de la desviación estándar. Otra forma es restringir el denominador para ser al menos ϵ.

Dada una imagen de entrada $\mathtt{X}$, GCN produce una imagen de salida $\mathtt{X}'$, definida de tal manera que:

$$\mathtt{X}'_{i,j,k} = s\frac{X_{i,j,k} - \bar{X}}{max\left\{\epsilon, \sqrt{\lambda + \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(X_{i,j,k} - \bar{X})^2}\right\}}$$

Aunque no es una desventaja quizá si que se ha visto necesario mejorar la normalización del contraste global ya que zonas como los bordes o las esquinas de las imágenes no suelen resaltarse y puede interesarnos. Esto hace que surja la normalización del contraste local (LCN). Esta normalización, en cambio, garantiza que el contraste se normalice de forma local, es decir, en cada ventana pequeña, en lugar de sobre la imagen en sus conjunto.

En la siguiente imagen podemos ver una comparación entre la normalización de contraste global y local.

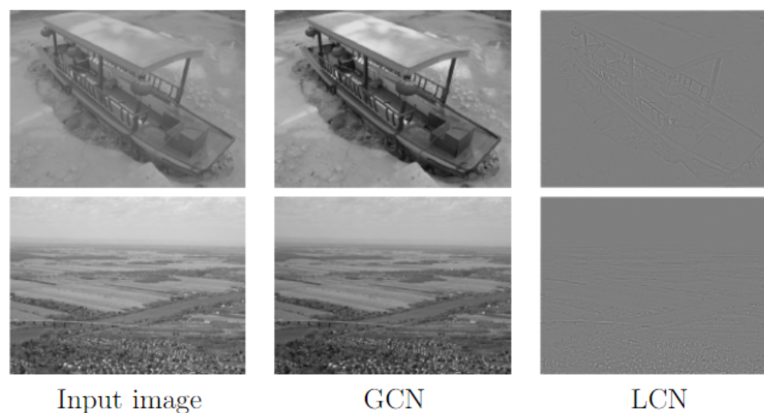


Figura 5.2: Comparación normalización de contraste global y local

Visualmente los efectos de la normalización del contraste global son sutiles ya que coloca todas las imágenes aproximadamente en la misma escala, lo que reduce la carga sobre el algoritmo de aprendizaje para manejar múltiples escalas. La normalización del contaste local modifica la imagen mucho más, descartando todas las regiones de intensidad constante. Esto permite que el modelo se centre solo en los bordes.

La normalización de contraste local es una operación diferenciable por lo que se puede aplicar a la oculta de la red como un efecto no lineal, así como a la operación de preprocesamiento de entrada.

5.3. Aumento del conjunto de datos de entrada

Es posible aumentar de una manera fácil el conjunto de datos de entrenamiento. Para ello se pueden agregar copias adicionales del conjunto de entrenamiento para mejorar así la generalización del clasificador. Estas copias se suelen general realizando ciertos cambios en la imagen original pero sin que cambie su categorización inicial. Centrándonos en la clasificación del reconocimiento de objetos, esta práctica es bastante adecuada ya que la información sobre la categoría es invariable para muchas de las tranformaciones. Algunas de estas transformaciones pueden ser rotación aleatoria de la imagen, deformación geométrica no lineal o la perturbación aleatorio de los colores de la imagen.

5.4. Aplicación

Con la finalidad de comprobar la utilidad y eficacia del Deep Learning para le procesamiento y reconocimiento de imágenes se ejecutará una aplicación en R donde se implementará un modelo capaz de clasificar imágenes.

5.4.1. Set de datos

Para llevar a cabo esta aplicación se usará la base de datos MNIST. Esta es una gran base datos de dígitos escritos a manos. Este set de datos contiene 60.000 imágenes de entrenamiento y 10.000 imágenes de prueba.

Posteriormente probaremos el clasificador pero esta vez usando número escritos manualmente en el programa Paint con el ratón.

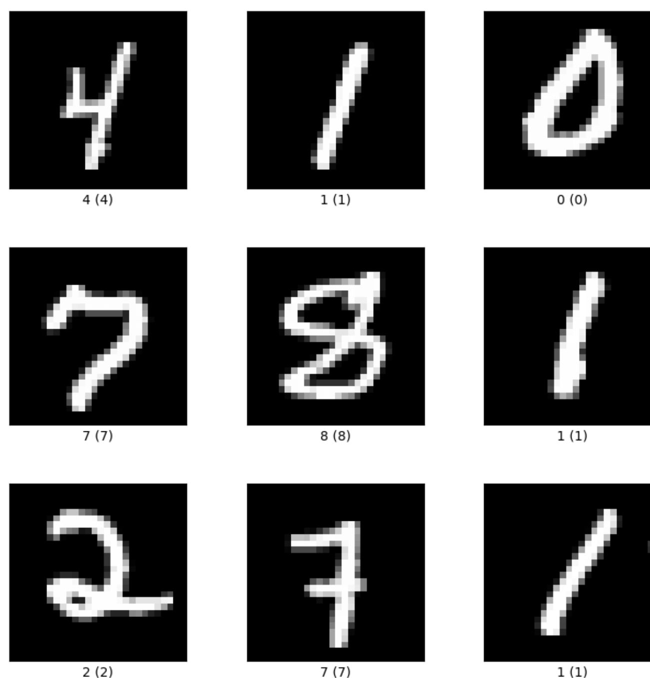


Figura 5.3: Ejemplo del conjunto de datos

5.4.2. Aplicación en R

5.4.2.1. Introducción a R

R es un lenguaje de programación interpretado, de distribución libre, bajo Licencia GNU principalmente utilizado para el cómputo estadístico y gráfico. Realmente es tanto un lenguaje como un sistema ya que esa era la idea de sus creadores, Robert Gentleman y Ross Ihaka.

R fue creado en 1992 en Nueva Zelanda por los mencionados ya Robert Gentleman y Ross Ihaka. La intención inicial con respecto a R era hacer un lenguaje didáctico para

ser utilizado en una asignatura en la Universidad de Nueva Zelanda y es por eso que adoptaron la sintaxis del lenguaje S. La semántica se diferencia del lenguaje S en detalles un poco más profundos de programación. El nombre de R tiene su origen en las iniciales de sus creadores.

En general, el sistema R está dividido en dos partes. Una primera parte compuesta por el sistema base de R y una segunda parte que incluye todo lo demás. La funcionalidad de R consta de paquetes modulares. Por defecto, en el sistema está incluído un paquete básico que contiene lo necesario para su ejecución y la mayoría de las funciones fundamentales.

Con respecto a los gráficos, R cuenta con muchos paquetes para manejar los datos y poder graficarlos, convirtiéndolo así en uno de los sistemas más sofisticados dentro de los paquetes estadísticos. Existen más de 4000 paquetes además del paquete base. Estos se pueden descargar en (http://cran.r-project.org), aunque también es posible instalarlos directamente desde R. Estos paquetes han sido desarrollados por usuarios y programadores alrededor del mundo. Además existen otros que están disponibles en redes personales y cuyos usuarios en ocasiones los comparten en repositorios públicos.

R es muy útil para el trabajo interactivo, pero también es un poderoso lenguaje de programación para el desarrollo de nuevas herramientas. Otra ventaja muy importante es que tiene una comunidad muy activa, por lo que, haciendo las preguntas correctas rápidamente encontrarás la solución a los problemas que se te presenten en el ámbito de la programación con R.

Una de las principales desventajas que tiene R es que debido a su estructura, consume mucha memoria y en ocasiones no es capaz de procesar datos de tamaño muy grande.

Las ventajas de R Software para la ciencia de datos:

- Algunas técnicas avanzadas y robustas son más óptimas con este software.
- El ambiente de trabajo es muy flexible y extensible.
- Se pueden crear gráficos de alta calidad y exportarlos en diferentes formatos.
- Consume pocos recursos informáticos.
- Se pueden crear aplicaciones web interactivas (apps) con la herramienta Shiny.
- Se puede crear un flujos de trabajo para escribir informes reproductibles y dinámicos.

5.4.2.2. Librerías utilizadas

A continuación se describen las librerias de R que principalmente se han utilizado en la aplicación.

- Tensorflow: es una librería de código abierto para realizar cálculos numéricos mediante diagramas de flujo de datos. Con Tensorflow lo que se codifica es un grafo cuyos nodos y aristas representan tensores (matrices de datos multidimensionales).Una de las ventajas de TensorFlow es la posibilidad de ejecutar en una o varias CPU o GPU en un PC o servidor.
- keras: es una 'API' de redes neuronales de un alto nivel. Fue desarrollado con el objetivo de una experimentación rápida. Admite tanto redes recurrentes como convolucionales, e incluso una combinación de ambas y se ejecuta sin problemas tanto en dispositivos de CPU como de GPU.

- dplyr: es un paquete que proporciona herramientas para poder manipular de una manera bastante eficiente conjuntos de datos.
- reticulate: es un paquete que nos permite llamar a Python desde R. Cuando se utiliza los tipos de datos R se convierten en sus equivalentes de Python y viceversa cuando se devuelven los valores.
- png: Este paquete nos proporciona una manera fácil y sencilla de leer y mostrar imágenes de mapa de bits almacenadas en formato PNG.

5.4.2.3. Aplicación

A continuación se muestra el código fuente de la aplicación.

En primer lugar cargamos todas las librerias necesarias:

```
library(tensorflow)
library(keras)
library(reticulate)
library(png)
```

El paquete 'keras' incluye la base de datos Mnist con la que se va a realizar el entrenamiento del modelo. Para descargarnos los datos, alojados en https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz usamos la función dataset_mnist().

```
dataset_inicial <- dataset_mnist()

#'Veamos qué hay dentro del set:
names(dataset_inicial)
```

```
## [1] "train" "test"
```

```
#'Además comprobemos qué hay dentro de train y de test:
names(dataset_inicial$train)
```

```
## [1] "x" "y"
```

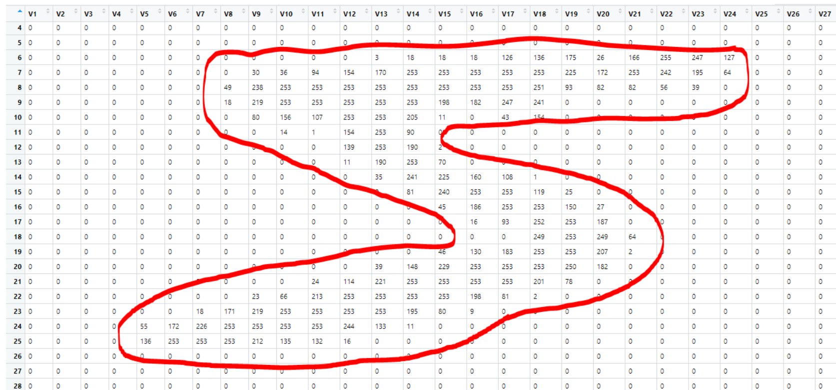
```
names(dataset_inicial$test)
```

```
## [1] "x" "y"
```

```
#'Los conjuntos 'y' corresponden con las etiquetas de los datos,
#'necesarias para un modelo supervisado.

#'Vamos a crear variables con los set de train y test,
#'tanto de los datos como de las etiquetas de los mismos:
train_set <- dataset_inicial$train$x
train_label <- dataset_inicial$train$y
test_set <- dataset_inicial$test$x
test_label <- dataset_inicial$test$y
```

El objeto train_set contiene la información de 6000 imágenes de 28x28. Veamos el ejemplo del primer número del set de entrenamiento.



	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	V25	V26	V27
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255	247	127	0	0	0
7	0	0	0	0	0	0	0	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242	195	64	0	0	0
8	0	0	0	0	0	0	0	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56	39	0	0	0	0
9	0	0	0	0	0	0	0	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	14	1	154	253	90	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	139	253	190	2	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	11	190	253	70	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	35	241	225	160	108	1	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	93	252	253	187	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	249	253	249	64	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	18	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	55	172	226	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	136	253	253	253	212	135	132	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 5.4: Primer número del conjunto de datos

En esta matriz de 28 x 28 cuanto más alto es el valor del número más oscura es la zona de la imagen. Cuando el valor es 0 es que no se escribió nada.

```r
#'A continuación vamos a redefinir las dimensiones de los datos.
#'Lo primero que vamos a hacer va a ser transformar la matriz de
#'28x28 por un vector de 1x784.

#'Para ello usaremos la funcion array_reshape() del paquete 'reticulate':
train_set <- array_reshape(train_set, c(nrow(train_set), 28, 28, 1))
test_set <- array_reshape(test_set, c(nrow(test_set), 28, 28, 1))
input_shape <- c(28, 28, 1)

#'Después hay que escalar los valores de dentro que van de 0 a 255
#'a valores entre 0 y 1.
train_set <- train_set / 255
test_set <- test_set / 255

#'Otro paso necesario es convertir la clase de las etiquetas en binario
#'con la función to_categorical esta vez del propio paquete 'keras'
train_label <- to_categorical(train_label, 10)
test_label <- to_categorical(test_label, 10)
```

Ahora que ya hemos realizado el pretratamiento de datos, vamos a definir el modelo que queremos entrenar, añadiendole tantas capas como se vean necesarias.

```r
#'La función keras_model_sequential sirve para definir un modelo
#'en 'keras' donde se enumeran las capas de las que se compone
#'el modelo.

model <- keras_model_sequential() %>%

  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu',
                input_shape = input_shape) %>%

  #' 'layer_conv_2d' es una capa de convolución. Añadiremos 2,
  #ambas con función de activación ReLu, tamaño del kernel 3x3.
```

```
  layer_conv_2d(filters = 64, kernel_size = c(3,3),
                activation = 'relu') %>%

  #'Aplicamos una capa de reducción o pooling:

  layer_max_pooling_2d(pool_size = c(2, 2)) %>%

  #' La siguiente capa tiene como finalidad evitar el sobreajuste:
  layer_dropout(rate = 0.25) %>%

  #' Esta capa siguiente aplana las entradas,
  #'sin afectar al tamaño total.

  layer_flatten() %>%

  #'Agregamos una capa totalmente concectada a una salida:

  layer_dense(units = 128, activation = 'relu') %>%

  #'Aplicamos otra capa para evitar el sobreajuste:

  layer_dropout(rate = 0.5) %>%

  #'Por último una capa totalmente conectada esta vez con dimensión
  #'10 de salida y función de activación softmax:
  layer_dense(units = 10, activation = 'softmax')
```

```
Epoch 1/12
375/375 [==============================] - 266s 709ms/step - loss: 0.2730 - accuracy: 0.9172 - val_loss:
 0.0728 - val_accuracy: 0.9779
Epoch 2/12
375/375 [==============================] - 316s 841ms/step - loss: 0.1008 - accuracy: 0.9696 - val_loss:
 0.0541 - val_accuracy: 0.9837
Epoch 3/12
375/375 [==============================] - 309s 824ms/step - loss: 0.0721 - accuracy: 0.9775 - val_loss:
 0.0463 - val_accuracy: 0.9875
Epoch 4/12
375/375 [==============================] - 310s 826ms/step - loss: 0.0589 - accuracy: 0.9819 - val_loss:
 0.0443 - val_accuracy: 0.9867
Epoch 5/12
375/375 [==============================] - 311s 829ms/step - loss: 0.0514 - accuracy: 0.9849 - val_loss:
 0.0388 - val_accuracy: 0.9887
Epoch 6/12
375/375 [==============================] - 311s 829ms/step - loss: 0.0438 - accuracy: 0.9867 - val_loss:
 0.0404 - val_accuracy: 0.9893
Epoch 7/12
375/375 [==============================] - 316s 843ms/step - loss: 0.0394 - accuracy: 0.9886 - val_loss:
 0.0406 - val_accuracy: 0.9893
Epoch 8/12
375/375 [==============================] - 311s 830ms/step - loss: 0.0348 - accuracy: 0.9893 - val_loss:
 0.0409 - val_accuracy: 0.9902
Epoch 9/12
375/375 [==============================] - 305s 813ms/step - loss: 0.0308 - accuracy: 0.9906 - val_loss:
 0.0375 - val_accuracy: 0.9898
Epoch 10/12
375/375 [==============================] - 309s 824ms/step - loss: 0.0290 - accuracy: 0.9913 - val_loss:
 0.0398 - val_accuracy: 0.9902
Epoch 11/12
375/375 [==============================] - 304s 811ms/step - loss: 0.0274 - accuracy: 0.9917 - val_loss:
 0.0361 - val_accuracy: 0.9903
Epoch 12/12
375/375 [==============================] - 305s 813ms/step - loss: 0.0275 - accuracy: 0.9915 - val_loss:
 0.0350 - val_accuracy: 0.9902
```

```
#' Comipilamos y entrenamos el modelo:

model %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
```

```
  metrics = c('accuracy')
)
set.seed(2611)

model %>% fit(
  train_set, train_label,
  batch_size = 128,
  epochs = 12,
  validation_split = 0.2
)

summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## conv2d (Conv2D)                     (None, 26, 26, 32)              320
## ________________________________________________________________________________
## conv2d_1 (Conv2D)                   (None, 24, 24, 64)              18496
## ________________________________________________________________________________
## max_pooling2d (MaxPooling2D)        (None, 12, 12, 64)              0
## ________________________________________________________________________________
## dropout (Dropout)                   (None, 12, 12, 64)              0
## ________________________________________________________________________________
## flatten (Flatten)                   (None, 9216)                    0
## ________________________________________________________________________________
## dense (Dense)                       (None, 128)                     1179776
## ________________________________________________________________________________
## dropout_1 (Dropout)                 (None, 128)                     0
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 10)                      1290
## ================================================================================
## Total params: 1,199,882
## Trainable params: 1,199,882
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
#Evaluamos ahora el modelo:
scores <- model %>% evaluate(
  test_set, test_label, verbose = 0
)

cat('Test accuracy:', scores[[2]], '\n')
```

```
## Test accuracy: 0.9916
```

Una vez que tenemos el modelo entrenado con una precisión suficientemente alta, vamos a terminar de validarlo usando esta vez imágenes escritas a mano. Para ello con el programa

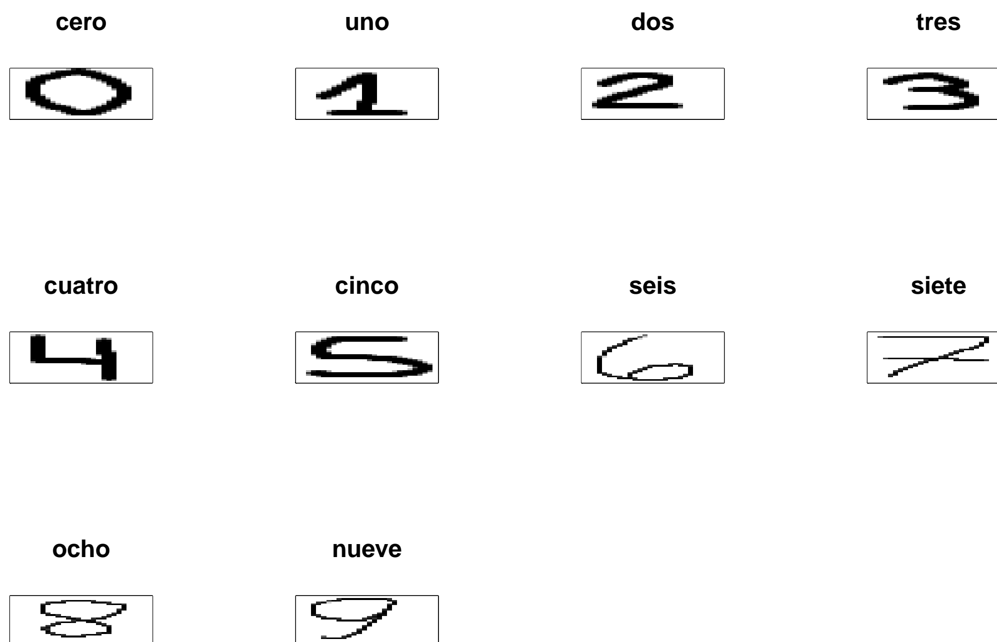
Paint escribimos los número del 0 al 9 ajustando el tamaño a 28x28 pixeles.

```
#'Para leer las imágenes utilizamos la función readPNG del paquete
#''png', datos que extraemos en una matriz directamente.

#'Vamos a crear una lista con todas las imágenes dibujadas previamente.
numeros <- list()
nombres <- c("cero", "uno", "dos", "tres", "cuatro", "cinco", "seis",
             "siete", "ocho", "nueve")

for (i in 1:10) {
  n<- readPNG(paste0("numeros/",nombres[i], ".png"))
  numeros[[i]] <- n[, , 1]
}

#'A continuación dibujemos estas imágenes
par(mfrow=c(3, 4))
for (j in 1:10) {
  ima <- numeros[[j]]
  ima <- abs(1-ima)
  ima <- 1 - t(apply(ima, 2, rev))
  image(1:28, 1:28, ima, col=gray((0:255)/255), xaxt='n', yaxt='n',
        xlab='', ylab='', main=nombres[j])
}
```




```
#'El siguiente paso es predecir con el modelo entrenado, a qué número
#'corresponde y compararlo con el número real.
pred=c()
```

```
for (i in 1:10) {
  #'La funcion readPNG lee una una imagen en una matriz
  n <- readPNG(paste0("numeros/",nombres[i], ".png"))
  n <- n[, , 1]
  n <- abs(1-n)
  n<-array_reshape(n, c(1, 28, 28, 1))
  prediccion <- model %>% predict_classes(n)
  pred=c(pred,prediccion)
}
real_label <- 0:9
cbind(real_label, pred)
```

```
##       real_label pred
##  [1,]          0    0
##  [2,]          1    1
##  [3,]          2    2
##  [4,]          3    3
##  [5,]          4    4
##  [6,]          5    5
##  [7,]          6    6
##  [8,]          7    4
##  [9,]          8    8
## [10,]          9    1
```

Podemos comprobar cómo ha predicho bien todo excepto el 7 y el 9.

Bibliografía

[1] Berzal, F. 2018. *Redes neuronales & deep learning*. Edición independiente. ISBN:1-0903-2030-2.

[2] Cruz, P.P. 2010. *Inteligencia artificial con aplicaciones a la ingeniería*. Alfaomega Grupo Editor, S.A. de C.V., México. ISBN: 978-607-7854-83-8.

[3] François Cholet, J.A. 2017. *Deep learning with r*. MEAP Edition. Manning Early Access Program.

[4] Giuseppe Ciaburro, B.V. 2017. *Neural networks with r: Smart models using cnn, rnn, deep learning, and artificial intelligence principles*. Packt. ISBN: 978-1-78839-787-2.

[5] Julio Sergio Santana, E.M.F. 2014. *El arte de programar en r: Un lenguaje para la estadística*. Instituto Mexicano de Tecnología del Agua. ISBN: 978- 607-9368-15-9.

[6] Luque-Calvo, P.L. 2017. *Escribir un trabajo fin de estudios con r markdown*. Disponible en http://destio.us.es/calvo.

[7] Shanmugamani, R. 2018. *Deep learning for computer vision: Expert techniques to train advanced neural networks using tensorflow and keras*. Packt. ISBN: 978-1-78829-562-8.

[8] Team, R.D.C. 2000. "Introducción a r: Un entorno de programación para análisis de datos y gráficos". Disponible en https://cran.r-project.org/doc/contrib/R-intro-1.1.0-espanol.1.pdf.

[9] Venkatesan, R. and Li, B. 2018. *Convolutional neural networks in visual computing: A concise guide*. CRC Press. ISBN: 978-1-4987-7039-2.

[10] "TensorFlow for r: Convolutional neural network (cnn)". Disponible en https://tensorflow.rstudio.com/tutorials/advanced/images/cnn/.