



# DEEP LEARNING

Isaac Pérez Borrero  
Manuel E. Gegúndez Arias



# DEEP LEARNING

## Fundamentos, teoría y aplicación



# DEEP LEARNING

## Fundamentos, teoría y aplicación

Isaac Pérez Borrero  
Manuel Emilio Gegúndez Arias



---

DATOS EDICIÓN

---

PRIMERA EDICIÓN EN FORMATO EBOOK: ABRIL 2021

PRIMERA EDICIÓN EN FORMATO PAPEL: ABRIL 2021

© Servicio de Publicaciones  
Universidad de Huelva

© Isaac Pérez Borrero 

© Manuel Emilio Gegúndez Arias 

I.S.B.N. (papel): 978-84-18628-28-3

E.I.S.B.N. (epub): 978-84-18628-29-0

Depósito legal: H 60-2021

---

PAPEL

---

*Papel*

Cartulina gráfica 300 g / Estucado mate 120 g

*Encuadernación*

Encuadernación PUR

Printed in Spain. Impreso en España.

*Maquetación y Ebook*

José Antonio Salas Hernández

---

Salvo indicación contraria, todos los contenidos de la edición electrónica se distribuyen bajo una licencia de uso y distribución "Creative Commons Reconocimiento, no comercial, compartir igual 4.0 Internacional" (CC BY-NC-SA 4.0).



---

CEP

---

Pérez Borrero, Isaac

Deep learning : fundamentos, teoría y aplicación / Isaac Pérez Borrero, Manuel Emilio Gegúndez Arias. -- Huelva: Universidad de Huelva, 2021

265 páginas ; 24 cm. – (Serie Alonso Barba ; 19)

ISBN (papel) 978-84-18628-28-3

ISBN (epub) 978-84-18628-29-0

1. Inteligencia artificial. – 2. Aprendizaje automático. -- I. Gegúndez Arias, Manuel Emilio. – II. González Galán, María Dolores. -- III. Universidad de Huelva. – VI. Título. – V. Serie 681.324/.325

---

Publicaciones de la Universidad de Huelva es miembro de UNE

---

EL EBOOK LE PERMITE

---



Citar el libro



Navegar por marcadores e hipervínculos



Realizar notas y búsquedas internas



Volver al índice pulsando el pie de la página



Comparte  
#LibrosUHU



Únete y comenta



Novedades a golpe de clic



Nuestras publicaciones en movimiento



Suscríbete a nuestras novedades

# Índice

Prefacio .....	19
Agradecimientos.....	21
Introducción .....	23
1. Redes neuronales.....	25
1.1. Introducción .....	25
1.2. Perceptrón.....	27
1.2.1. Funciones de activación .....	33
1.2.2. Principales aplicaciones: clasificación y regresión.....	50
1.2.3. Limitaciones .....	54
1.3. Redes neuronales.....	64
1.3.1. Arquitectura.....	64
1.3.2. Aplicaciones .....	76
1.3.3. Limitaciones .....	79
2. Desarrollo de redes neuronales .....	89
2.1. Etapa de entrenamiento vs. inferencia.....	90
2.2. Paradigmas de aprendizaje .....	91
2.3. Obtención y preparación de los datos.....	95
2.4. Métricas de evaluación .....	98
2.5. Estimación del error .....	106
2.6. Aprendizaje basado en el gradiente .....	109
2.6.1. Inicialización de los pesos.....	121
2.6.2. Principales algoritmos de optimización .....	124
2.6.3. Backpropagation .....	135
2.6.4. Problemas del aprendizaje basado en gradiente.....	148

2.7. Parámetros del entrenamiento .....	156
2.8. Control y seguimiento del entrenamiento .....	163
3. Redes neuronales recurrentes .....	177
3.1. Introducción .....	177
3.2. Redes neuronales recurrentes .....	180
3.2.1. Entrenamiento de una red neuronal recurrente .....	184
3.2.2. Aplicaciones .....	188
3.2.3. Limitaciones .....	189
4. Deep learning .....	191
4.1. Introducción .....	191
4.2. Redes neuronales convolucionales .....	203
4.2.1. Arquitectura.....	212
4.2.2. Aplicaciones .....	237
4.2.3. Arquitecturas populares.....	249
Bibliografía.....	257

# Índice de figuras

1.1. Evolución del ratio entre el número de publicaciones sobre conexiónismo y simbolismo .....	27
1.2. Partes de una neurona biológica .....	28
1.3. Neurona de McCulloch-Pitts.....	29
1.4. Interpretación geométrica de un perceptrón que modela la función OR.....	31
1.5. Interpretación geométrica de un perceptrón que modela la función OR con tres variables .....	34
1.6. El perceptrón.....	35
1.7. El vector normal al hiperplano definido por los pesos del perceptrón .....	37
1.8. Perceptrón (sin bias) para decidir si comprar o no una casa.....	38
1.9. Ejemplos de hiperplanos en función del valor de $w_1$ y $w_2$ (con $w_0 = 0$ ).....	39
1.10. División del espacio para diferentes hiperplanos de ejemplo .....	40
1.11. Perceptrón para decidir si comprar o no una casa a partir de $100m^2$ .....	41
1.12. Perceptrón con dos entradas ( $x_1$ y $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación identidad .....	45
1.13. Perceptrón con dos entradas ( $x_1$ y $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación escalonada .....	46
1.14. Perceptrón con dos entradas ( $x_1$ y $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación sigmoide.....	47
1.15. Perceptrón con dos entradas ( $x_1$ y $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación $\tanh$ .....	47
1.16. Perceptrón con dos entradas ( $x_1$ y $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación ReLU .....	49
1.17. Clasificación multiclasa con el enfoque uno contra todos .....	52
1.18. Hiperplano (línea punteada) en un problema de clasificación (izquierda) y regresión (derecha) .....	53
1.19. Ejemplo de separación lineal mediante un hiperplano para la función OR y AND .....	55
1.20. Ejemplo de outlier en los datos.....	57
1.21. Ejemplo de datos no separables linealmente .....	57

1.22. Función XOR .....	58
1.23. Salida de perceptrón que aproxima la función OR.....	59
1.24. Salida de perceptrón que aproxima la función NOT AND.....	59
1.25. Salida de perceptrón que aproxima la función AND.....	60
1.26. Tres perceptrones aproximando la función XOR.....	60
1.27. Salida de un perceptrón que aproxima la función OR .....	61
1.28. Salida de un perceptrón que aproxima la función NOT AND.....	61
1.29. Salida construida con tres perceptrones para aproximar la función XOR .....	62
1.30. Capa de una red neuronal.....	65
1.31. Elementos de una red neuronal .....	73
1.32. Red neuronal para aproximar la función XOR.....	75
1.33. Transformación softmax y codificación one-hot.....	77
1.34. Ejemplo de la operación flattening .....	78
1.35. Efecto del cambio del valor del peso y del bias en una neurona con función de activación sigmoide.....	83
1.36. Construcción de una función «pulso» con una red neuronal.....	84
1.37. Construcción de una función «pulso» ampliado con una red neuronal .....	85
1.38. Aproximación de una función con una red neuronal .....	86
1.39. Ejemplo de función aproximada correctamente solo en algunas zonas.....	87
 2.1. Etapa de entrenamiento vs. inferencia.....	91
2.2. Aprendizaje supervisado, no supervisado y por refuerzo.....	94
2.3. Efecto en la clasificación de diferentes valores de umbrales .....	100
2.4. Matriz de confusión para N clases .....	102
2.5. Matriz de confusión considerando solo las clases positivas y negativas .....	103
2.6. Métricas para un problema de clasificación de personas según están embarazadas o no.....	104
2.7. Curva ROC.....	105
2.8. Métrica IoU .....	106
2.9. Gráfica de la función ECM .....	108
2.10. Gráfica de la función CE .....	109

2.11. Superficie de la función de pérdida para un perceptrón con dos entradas.....	111
2.12. Superficie típica de la función de pérdida de una red neuronal .....	112
2.13. Evolución del vector de pesos durante el algoritmo de entrenamiento del perceptrón.....	115
2.14. Representación gráfica de una función y su derivada.....	116
2.15. Posibles puntos críticos de una función .....	117
2.16. Efecto en la definición de la superficie de error de las diferentes estrategias de optimización .....	120
2.17. Regiones de la superficie del error ideales para comenzar el entrenamiento .....	122
2.18. Modificación de la variable para acercarse al mínimo de la función.....	126
2.19. Modificación de los pesos por el algoritmo descenso por gradiente .....	127
2.20. Comparativa del algoritmo de descenso por gradiente con y sin momento .....	128
2.21. Función de dos variables y plano de sus curvas de nivel .....	129
2.22. Reducción de la oscilación por el uso del momento .....	130
2.23. Ejemplo de iteraciones del algoritmo descenso por gradiente .....	132
2.24. Ejemplo de grafo de operaciones .....	142
2.25. Ejemplo completo de red neuronal para el problema XOR .....	143
2.26. Grafo de operaciones para la red neuronal de la Figura 2.25 .....	146
2.27. Cálculo de una iteración del algoritmo descenso por gradiente sobre el grafo de operaciones de una red .....	147
2.28. Gráfica de las derivadas de las principales funciones de activación.....	fun- ciones 149
2.29. Ejemplo de sobreajuste (izquierda), buen ajuste (centro) y infraajuste (derecha) en base a un conjunto de datos de ejemplo (color morado) .....	152
2.30. Posibles escenarios de un modelo respecto al sesgo y la varianza .....	al sesgo 154
2.31. Búsqueda de compromiso entre sesgo y rianza al definir la complejidad del modelo .....	va- rianza 155
2.32. Ciclo de vida del desarrollo de redes neuronales.....	157

2.33. Relación entre los datos de un conjunto de datos, el tamaño del lote, la iteración y la época de entrenamiento.....	158
2.34. Diferencias entre un mínimo pronunciado y aplanado de una función de pérdida definida por un lote de ejemplos y la definida por todos los datos .....	161
2.35. Gráficas habituales del entrenamiento de una red neuronal.....	165
2.36. Detección de sobreajuste e infraajuste en la gráfica de pérdida .....	166
2.37. Efecto del valor de learning rate en la gráfica de pérdida .....	167
2.38. Diferencias en el descenso por gradiente con un valor para el learning rate fijo y variable.....	169
2.39. Superficie de la función de pérdida con datos normalizados y sin normalizar .....	170
2.40. Efecto de la regularización en la complejidad del modelo .....	171
2.41. Red con y sin dropout .....	173
2.42. Efecto de la limitación del gradiente durante el entrenamiento .....	175
 3.1. Sin información de los instantes anteriores, no se puede predecir la siguiente posición del vagón.....	178
3.2. En base a la información anterior, se puede predecir que el vagón se moverá hacia la derecha.....	179
3.3. Esquema de una red neuronal recurrente.....	181
3.4. Esquema de una red neuronal recurrente multicapa.....	183
3.5. División en ventanas de la secuencia de datos y uso en cada iteración de entrenamiento para entrenar (E), validar (V) y pruebas (P) .....	186
3.6. Proceso de desenrollado del grafo de operaciones para dos iteraciones de una red neuronal recurrente .....	187
 4.1. Relación entre inteligencia artificial, machine learning y deep learning.....	ma- 191
4.2. Comparativa entre el desarrollo de un sistema basado en técnicas de machine learning y deep learning.....	193
4.3. Descomposición de los píxeles de una imagen en una nueva representación jerárquica de características de menor a mayor complejidad .....	195

4.4. Esquema completo de un modelo de deep learning con sus diferentes etapas.....	196
4.5. Evolución del porcentaje de error cometido por el modelo ganador de ImageNet a lo largo de los últimos años .....	197
4.6. Comparativa del rendimiento de los modelos de machine learning y deep learning en base al tamaño del conjunto de datos disponible .....	199
4.7. Coste del almacenamiento de los datos frente a su disponibilidad.....	200
4.8. De izquierda a derecha: Yann LeCun, Geoffrey Hinton y Yoshua Bengio.....	201
4.9. Panorama industrial del deep learning .....	202
4.10. Experimentos llevados a cabo por Hubel y Wiesel .....	204
4.11. Localización de la corteza visual en el cerebro .....	204
4.12. Jerarquía de características que obtiene el cerebro a partir de la información visual de la retina .....	208
4.13. Células S que comparten pesos para extraer la misma característica en diferentes regiones de la entrada.....	209
4.14. Estructura del Neocognitron .....	209
4.15. Características extraídas por la primera capa de una red neuronal convolucional. Arriba se muestra la imagen que produce mayor activación de la característica y, debajo, ejemplos de imágenes reales del conjunto de datos que producen una alta activación. ....	210
4.16. Características extraídas por la segunda capa de una red neuronal convolucional. A la izquierda se muestra la imagen que produce mayor activación de la característica y, a la derecha, ejemplos de imágenes reales del conjunto de datos que producen una alta activación. ....	211
4.17. Características extraídas por la tercera capa de una red neuronal convolucional. A la izquierda se muestra la imagen que produce mayor activación de la característica y, a la derecha, ejemplos de imágenes reales del conjunto de datos que producen una alta activación. ....	211
4.18. Ejemplos de diferentes datos, su representación y la denominación que tienen en base a su dimensión .....	213
4.19. Ejemplo gráfico del cálculo de la convolución de dos funciones continuas .....	221
4.20. Ejemplo gráfico del cálculo de la convolución de dos funciones discretas.....	222

4.21. Representación de la convolución de dos piezas de información de una dimensión con tamaño limitado y diferente .....	223
4.22. Representación de la convolución de dos piezas de información con relleno.....	224
4.23. Representación de la convolución de dos piezas de información con un paso de 2 .....	225
4.24. Representación de la convolución de dos piezas de información de dos dimensiones.....	226
4.25. Diferentes convoluciones sobre una imagen de tres dimensiones (canales rojo, verde y azul) .....	227
4.26. Información relativa a una capa de convolución .....	228
4.27. Comparativa entre una capa de una red neuronal y una capa de convolución .....	229
4.28. Mejora de los resultados con el aumento la profundidad de la red Error Top-5 es el error cometido por la red al no estar la clase esperada entre las 5 primeras clases del modelo. ....	230
4.29. Ejemplo del cálculo de max-pooling y average pooling .....	232
4.30. Ejemplo de bloque inception.....	234
4.31. Ejemplo de skip connection .....	235
4.32. Ejemplo de dense block.....	236
4.33. Principales casos de uso del deep learning.....	238
4.34. De izquierda a derecha: clasificación, de- tección y segmentación .....	239
4.35. Red de conducción autónoma de Tesla: HydraNet .....	239
4.36. Red neuronal convolucional y LSTM usadas para describir una imagen .....	240
4.37. Reescalado de imágenes .....	241
4.38. Reconstrucción y coloreado de imágenes .....	241
4.39. Representación invariante de objetos .....	242
4.40. Transferencia del estilo de una imagen a otra .....	242
4.41. Estimación de la profundidad.....	243
4.42. Estimación de la pose .....	243
4.43. Modelo basado en deep learning entrenado por aprendizaje por refuerzo para jugar a los juegos de Atari .....	244

4.44. Modelo basado en deep learning entrenado por aprendizaje por refuerzo para jugar al juego del Go.....	245
4.45. Conducción autónoma con modelo de deep learning entrenado con aprendizaje por refuerzo .....	245
4.46. Manipulación de objetos con modelo de deep learning entrenado por aprendizaje por refuerzo .....	246
4.47. Procesamiento del lenguaje haciendo uso de una red neuronal convolucional .....	246
4.48. Reconocimiento del audio haciendo uso de una red neuronal convolucional combinada con una red neuronal recurrente.....	247
4.49. Ejemplos de segmentaciones por instancias realizadas con la red Mask R-CNN.....	248
4.50. Arquitectura de la red LeNet-5 .....	250
4.51. Arquitectura de la red AlexNet .....	251
4.52. Arquitectura de la red GoogleNet.....	252
4.53. Arquitectura de las redes ResNet-50, ResNet-101 y ResNet-152 .....	254
4.54. Arquitectura de la red DenseNet-121 .....	256

## Índice de tablas

1.1. Salidas de una neurona de McCulloch-Pitts para el Ejemplo 1.1 con 3 entradas $y = 3$ .....	42
1.2. Principales funciones de activación .....	44
1.3. Funciones booleanas diferentes para dos variables.....	56
1.4. Funciones booleanas diferentes, separables linealmente y ratio en función del número de variables .....	56
1.5. Entradas y salidas de cada uno de los perceptrones de la red neuronal para aproximar la función XOR .....	63

## Índice de algoritmos

1. Algoritmo de entrenamiento del perceptrón .....	114
2. Algoritmos basados en el gradiente .....	124





## Prefacio

El estudio de una nueva materia nunca es fácil. En el caso del deep learning, además, hay que añadir su temprana edad y su carácter multidisciplinar, ya que engloba conceptos de diferentes campos como la matemática, la neurociencia y la informática.

Por estos motivos, este libro pretende ser una base para el estudio del deep learning en español, permitiendo adquirir todos los conceptos necesarios para su dominio en una única obra. Se intenta explicar de forma amena los pormenores matemáticos sin perder rigor, aunque desde una perspectiva divulgadora, con el objetivo de hacer más ameno el estudio.

Con este libro se pretende que el alumno sea capaz de solventar cualquier duda conceptual que pudiera surgirle durante el estudio de una asignatura relacionada con deep learning. Para ello, se hace hincapié en multitud de conceptos básicos que suelen ser ignorados en la mayoría de los libros de deep learning al considerarse parte de otras materias. Además, se han introducido multitud de ejemplos y figuras que ayudan a la comprensión de los conceptos más complejos.



## Agradecimientos

Nos gustaría agradecer a toda la comunidad de deep learning el acceso libre tanto al conocimiento como a la tecnología. A diferencia de otras disciplinas que se encuentran en sus comienzos, los recursos disponibles relacionados con deep learning abundan en la red de forma gratuita y desinteresada.

Sin la apuesta por una circulación libre de conocimiento no hubiera sido posible la revolución que ha supuesto el deep learning para la sociedad y, en última instancia, la creación de este libro.

Desde estas líneas invitamos a otros profesionales a seguir el mismo espíritu y compartir su conocimiento con el resto de la comunidad para, entre todos, seguir avanzando en un campo con tanto potencial como es el deep learning.

Si encuentra alguna errata, le agradeceríamos que nos escribiera un correo a la dirección [isaac.perez@dci.uhu.es](mailto:isaac.perez@dci.uhu.es).



## Introducción

La organización de este libro busca proveer al estudiante de los conocimientos básicos necesarios para adentrarse en el estudio del deep learning.

El primer capítulo se centra en el estudio de las redes neuronales, modelo sobre el que se ha desarrollado el campo del deep learning. El estudio de estas redes abarca desde su historia hasta los conceptos más elementales con los que guardan relación.

En el segundo capítulo se aborda el desarrollo de las redes neuronales, que es transversal a todos capítulos del libro y que provee de los conocimientos necesarios para desarrollar cualquier modelo de deep learning.

En el tercer capítulo se aborda el procesamiento de series temporales mediante el uso de redes neuronales. Este tipo de redes son necesarias para abordar problemas donde los datos no son independientes y presentan algún tipo de relación.

Por último, en el cuarto capítulo se profundiza en el deep learning. Se comienza con una pequeña introducción divulgativa para familiarizar al estudiante con esta nueva disciplina para, después, pasar a estudiar el principal modelo de deep learning, las redes neuronales convolucionales.



# Capítulo 1

## Redes neuronales

### 1.1. Introducción

Las redes neuronales son una de las técnicas más utilizadas del aprendizaje automático (en inglés, *machine learning*). Este campo de la inteligencia artificial engloba a aquellos algoritmos basados en modelos (representaciones matemáticas de la realidad) predefinidos que son ajustados en base a los datos disponibles para resolver un problema específico.

En los comienzos de la inteligencia artificial, los esfuerzos se centraban en el estudio de algoritmos que permitiesen a los ordenadores hacer las tareas que pueden hacer los humanos (como razonar, hablar, reconocer lo que ven, etc.). Para muchas de estas tareas ha sido posible encontrar algoritmos que las resuelvan. Este es el caso de la búsqueda del camino más corto, donde el algoritmo A\* [1] es un claro ejemplo de ello. Sin embargo, para la mayoría de los problemas que los humanos resuelven no existe, de forma evidente, un algoritmo que permita a los ordenadores resolverlo. Por ejemplo, decir si hay o no un determinado objeto en una imagen es una tarea que, hasta la fecha, ha demostrado ser extremadamente difícil para un ordenador.

Debido a esta limitación, comenzó a surgir interés en técnicas que permitieran obtener estos algoritmos sin necesidad de detallar los pasos que deben seguir, de forma similar a cómo los humanos aprenden ciertas tareas: mediante un proceso de aprendizaje en base a unos datos de ejemplo. Es aquí cuando aparece el *machine learning* que, a diferencia de las técnicas tradicionales de inteligencia artificial, es

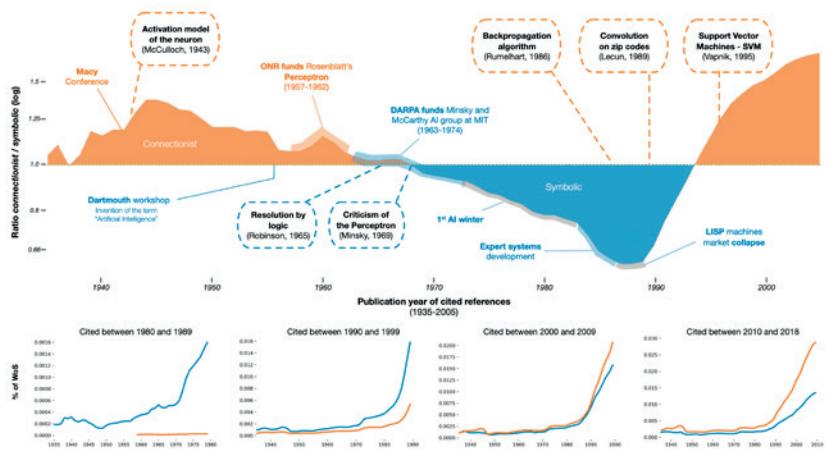
capaz de ajustar modelos prediseñados en base a los datos disponibles del problema para resolverlo.

Las técnicas o algoritmos de *machine learning* se diferencian en cómo definen este modelo. Por ejemplo, en el caso de las redes neuronales se utilizan modelos matemáticos inspirados en las neuronas. Principalmente, se distinguen dos tipos de paradigmas a la hora de definir estos modelos, el simbolismo y el conexionismo:

- El **simbolismo** se basa en el análisis formal. Se hace uso de la lógica para la manipulación de las representaciones abstractas del problema. Gracias a la naturaleza de estas técnicas, es posible saber en todo momento el porqué de la decisión tomada por el algoritmo, lo que permite tener más confianza en estas técnicas a la hora de su uso en problemas reales. Dentro del simbolismo, los modelos más usados son los sistemas basados en reglas y los árboles de decisión.
- El **conexionismo**, por el contrario, carece de esta calidad autoexplicativa de las técnicas anteriores. Se basa en la búsqueda de patrones en los datos para ser procesados mediante una red de nodos conectados donde su distribución y conexiones representan el modelo para el problema en cuestión. El principal modelo del conexionismo son las redes neuronales.

A lo largo de la historia, estos dos paradigmas han ido turnándose como el más prometedor, debido principalmente a las expectativas que se creaban con cada nuevo descubrimiento. Este hecho queda reflejado en el número de publicaciones relacionadas con cada paradigma a lo largo del tiempo (ver Figura 1.1). Como se aprecia en la Figura 1.1, el conexionismo es, actualmente, el paradigma que goza de una mayor popularidad. Esto se debe, en gran medida, a los resultados obtenidos por los modelos de *deep learning*, basados en las redes neuronales.

En la siguiente sección estudiaremos el principal componente de las redes neuronales: el **perceptrón**.



**Figura 1.1:** Evolución del ratio entre el número de publicaciones sobre conexiónismo (naranja) y simbolismo (azul), ambos normalizados por el número total de publicaciones indexadas en *Web of Science*

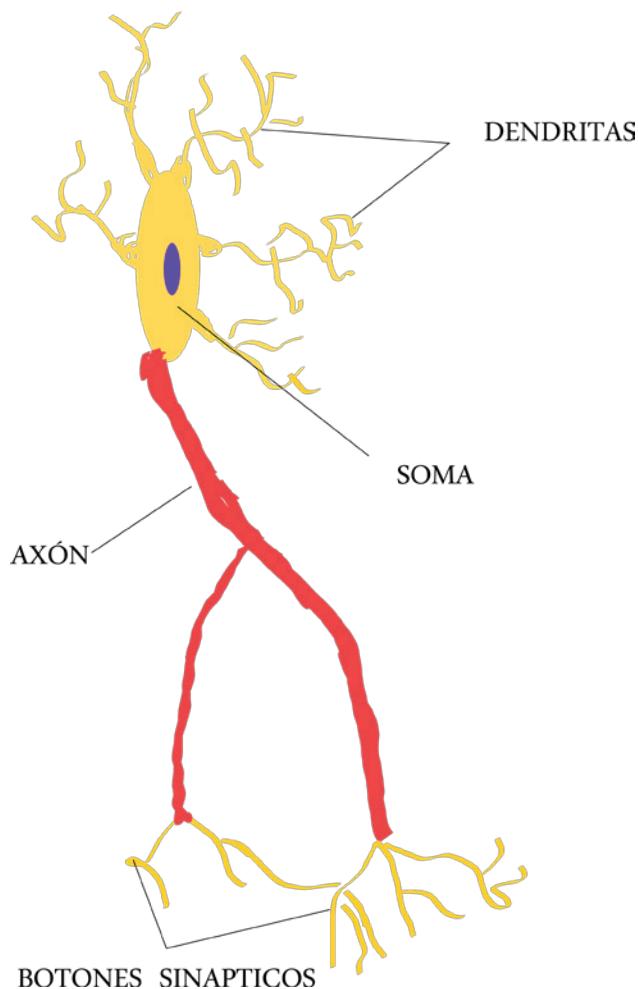
Fuente: <https://neurovenge.antonomase.fr/>

## 1.2. Perceptrón

Los orígenes del perceptrón datan del año 1943, mucho antes incluso de que se inventara el término «inteligencia artificial» (Conferencia de Dartmouth, en el año 1956). Fue en este año cuando Warren McCulloch y Walter Pitts, el primero neurócientífico y el segundo especialista en lógica, publicaron el primer modelo matemático de las neuronas del cerebro, conocido como la neurona de McCulloch-Pitts [2].

Este modelo matemático se inspira en la neurona biológica (ver Figura 1.2), que se compone de tres partes principales: el soma, las dendritas y el axón. El soma o cuerpo celular es el encargado de agregar los impulsos nerviosos provenientes de otras neuronas que llegan a través de los canales de entrada, las dendritas. Si al agregar estos impulsos se supera un cierto umbral, la neurona se activa y envía un impulso por el canal de salida, el axón. El axón se conecta a las dendritas de otras neuronas mediante lo que se conoce como conexión sináptica. Esta conexión no es completa, ya que existe una pequeña distancia entre el axón y la dendrita que per-

mite a la neurona regular el impulso que envía. La transmisión puede ser excitadora o inhibidora, lo que influye en la activación de las neuronas receptoras.



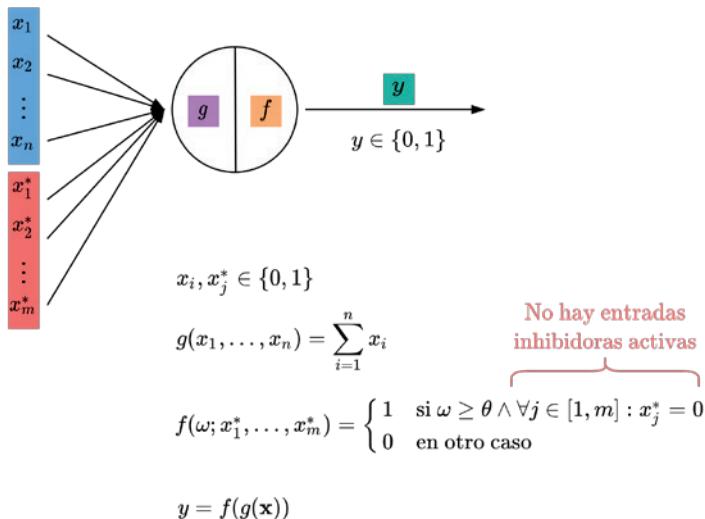
**Figura 1.2:** Partes de una neurona biológica

Fuente: [https://commons.wikimedia.org/wiki/File:Esquema\\_basico\\_de\\_una\\_neurona.svg](https://commons.wikimedia.org/wiki/File:Esquema_basico_de_una_neurona.svg)

Las neuronas son las unidades básicas de procesamiento de nuestro cerebro y sir-

vieron de inspiración a Warren McCulloch y Walter Pitts para el desarrollo de su neurona artificial. Al igual que en la neurona biológica, la neurona artificial de McCulloch y Pitts (ver Figura 1.3) recibe dos tipos de entradas (variables binarias): excitadoras ( $x_i \in \{0, 1\}$ ) e inhibidoras ( $x_j^* \in \{0, 1\}$ ). La neurona realiza la agregación de las entradas excitadoras mediante su suma y, cuando supera cierto umbral, permite la activación de la neurona, siempre y cuando no haya alguna entrada inhibidora activa.

En el caso de que la neurona se active, su salida valdrá uno; y cero si no se activa. Si alguna entrada inhibidora está activa, la salida valdrá cero, independientemente de que la agregación supere el valor de umbral o no.



**Figura 1.3:** Neurona de McCulloch-Pitts

En la Figura 1.3 se puede observar la analogía entre la neurona biológica y su modelo matemático. Vemos que las entradas (dendritas), en color azul y rojo, se representan como una lista de variables:  $[x_1, \dots, x_n]$  para las excitadoras y  $[x_1^*, \dots, x_k^*]$  para las inhibidoras. El cuerpo de la neurona (soma), representado como una cir-

cunferencia, recibe las entradas y realiza la agregación, en color morado, mediante la función  $g$ . También, en el cuerpo de la neurona, se define una función de activación, en color naranja,  $f$ , para decidir si la agregación ha activado o no a la neurona en base a que supere el umbral,  $\theta$ , y no haya entradas inhibidoras activas. Por último, el valor obtenido de la función de activación pasa al axón,  $y$ , que es la salida de la neurona, en color verde.

En esta neurona, tanto las entradas como las salidas son valores binarios. Por lo tanto, el principal uso de este modelo es la implementación de funciones booleanas. Mediante la definición del parámetro  $\theta$  es posible describir el comportamiento deseado de la neurona. A continuación, se muestra un problema de ejemplo en el que se podría emplear el modelo de neurona diseñado por McCulloch y Pitts.

**Ejemplo 1.1 :** Sistema de decisión para saber si debemos ir al cine a ver una película. Consideramos que el sistema debe tener en cuenta nuestro ánimo, si el género de la película es de aventuras y si actúa Johnny Depp. Por lo tanto, son necesarias tres variables de entrada:

- **estoyAlegre.** Variable que representa nuestro estado de ánimo. Vale 1 en el caso que estemos alegre y 0 en el caso contrario.
- **generoAventuras.** Esta variable indica si la película es del género aventuras. Si lo es, vale 1, si no, vale 0.
- **actuaJohnnyDepp.** Con esta variable se indica el actor Johnny Depp actúa en la película. Vale 1 en el caso de que actúe y 0 en el caso contrario.

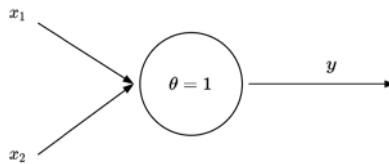
Solo iremos a ver la película si se cumplen las tres condiciones.

Para construir un modelo que resuelva el Ejemplo 1.1 con la neurona de McCulloch y Pitts hay que definir el parámetro  $\theta$ . Dada la condición de activación del modelo: «Solo iremos a ver la película si se cumplen las tres condiciones», el valor del umbral debe ser tres,  $\theta = 3$ , ya que este valor hace que el modelo se active solo si las tres entradas están activas. En la Tabla 1.1 se muestra la salida del modelo para todos los valores posibles de entrada (para este ejemplo, las tres entradas son excitadoras).

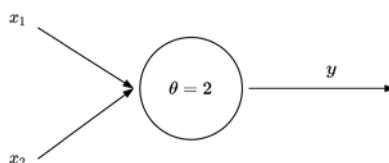
El perceptrón del Ejemplo 1.1 se corresponde con una función AND: la salida

solo estará activa si todas las entradas lo están. Con la neurona McCulloch-Pitts se pueden representar las funciones booleanas más comunes:

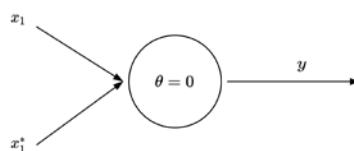
- **Función OR:**



- **Función AND:**



- **Función NOT** (se usa la propia entrada como entrada inhibidora):

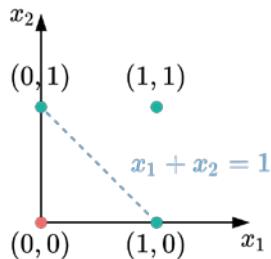


Otra forma de ver cómo funcionan estos modelos consiste en darle una interpretación geométrica. Cada una de las entradas del modelo se pueden ver como un punto del espacio de las variables de entrada. Este espacio tendrá tantas dimensiones como entradas haya. En el Ejemplo 1.1 el espacio tiene tres dimensiones

(es un cubo discreto no continuo). Lo que el modelo hace es dividir este espacio en dos partes mediante un hiperplano de separación, definido por la ecuación  $\sum_{i=1}^n x_i = \theta$ . Una de las partes en la que se divide el espacio la forman los puntos situados sobre o por encima del hiperplano, que provocan la activación del modelo. La otra parte la forman los puntos que se sitúan por debajo del hiperplano y, por lo tanto, no activan el modelo.

Este hiperplano será una línea para el caso particular de dos variables de entrada (dos dimensiones) o un plano si tenemos tres variables (tres dimensiones). De forma general se habla de hiperplano para no tener que hacer referencia a cada caso particular. También se puede llamar a este hiperplano frontera de decisión, ya que es donde cambia la salida del modelo de cero a uno o viceversa.

Tomando el modelo de la función OR con dos entradas ( $y = x_1 + x_2 \geq 1$ ), el hiperplano o frontera de decisión sería  $x_1 + x_2 = 1$ , que, en este caso, se corresponde con una línea al ser dos variables de entrada. En la Figura 1.4 se muestra la representación geométrica de este modelo.



**Figura 1.4:** Interpretación geométrica de un perceptrón que modela la función OR. En verde se indican las entradas que activan la salida y en rojo las que no. La frontera de decisión se marca con una línea punteada en color azul.

Dado que el modelo cuenta con dos variables binarias, el espacio de las variables de entrada solo está definido para los siguientes puntos:

- $x_1 = 0$  y  $x_2 = 0$ . En este caso la salida no se activa:

$$y = \sum_{i=1}^2 x_i \geq 1, \quad x_1 + x_2 = 0 + 0 = 0 \implies x_1 + x_2 \not\geq 1 \implies y = 0$$

- $x_1 = 0$  y  $x_2 = 1$ . En este caso la salida si se activa:

$$y = \sum_{i=1}^2 x_i \geq 1, \quad x_1 + x_2 = 0 + 1 = 1 \implies x_1 + x_2 \geq 1 \implies y = 1$$

- $x_1 = 1$  y  $x_2 = 0$ . En este caso la salida si se activa:

$$y = \sum_{i=1}^2 x_i \geq 1, \quad x_1 + x_2 = 1 + 0 = 1 \implies x_1 + x_2 \geq 1 \implies y = 1$$

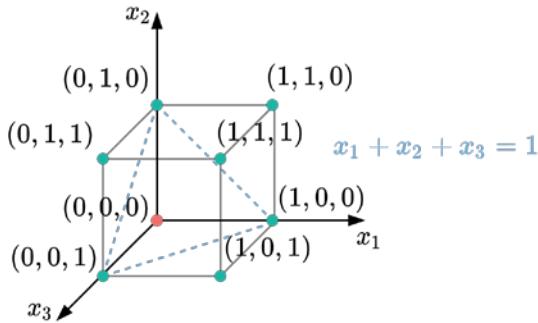
- $x_1 = 1$  y  $x_2 = 1$ . En este caso la salida si se activa:

$$y = \sum_{i=1}^2 x_i \geq 1, \quad x_1 + x_2 = 1 + 1 = 2 \implies x_1 + x_2 \geq 1 \implies y = 1$$

En la Figura 1.5 se muestra el hiperplano de separación para la función OR con tres variables de entrada (un plano en este caso).

Este primer modelo, aunque útil, se encuentra limitado a las funciones booleanas y no cuenta con un mecanismo para el ajuste de forma automática del valor de umbral,  $\theta$ , sino que tiene que definirse previamente. Además, puede que no interese que todas las entradas sean iguales y se quiera asignar más peso (importancia) a algunas entradas que a otras.

Para solventar las carencias que presentaba la neurona de McCulloch-Pitts, en 1958, Frank Rosenblatt presenta el *perceptrón* [3]. El perceptrón puede procesar cualquier entrada real, **ya no se limita a valores binarios**, lo que permite prescin-



**Figura 1.5:** Interpretación geométrica de un perceptrón que modela la función OR con tres variables. En verde se indican las entradas que activan la salida y en rojo las que no. La frontera de decisión se define con el plano que forman las tres líneas punteadas en color azul.

dir de las entradas inhibidoras, ya que los valores negativos de las entradas simulan el mismo comportamiento. Además, al estar cada entrada ponderada por un peso, se puede controlar la importancia de estas para el modelo.

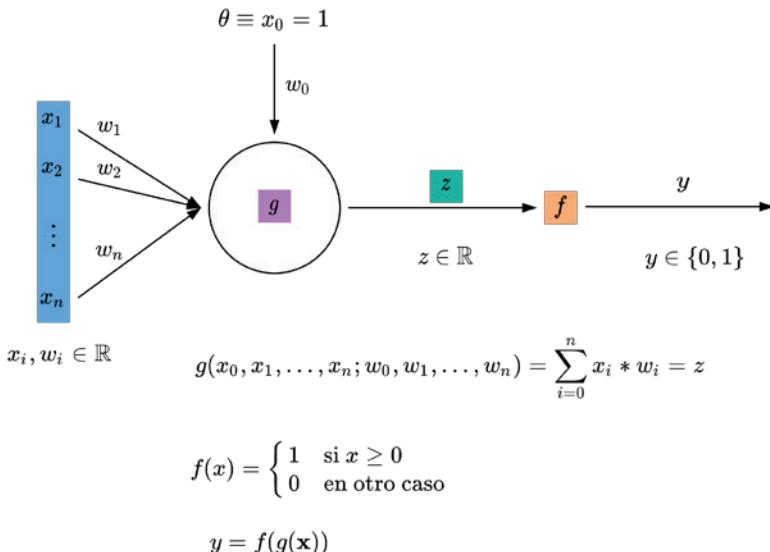
El ajuste del umbral se consigue añadiendo al perceptrón una nueva entrada con el valor de umbral, que, al contar con un peso, permite ajustarlo. Geométricamente, permite al perceptrón trazar el hiperplano de separación sin necesidad de pasar por el origen, gracias a añadir un término independiente a su definición: el peso del umbral, que actúa como un sesgo que se introduce en el sistema.

Otra de las novedades es que se elimina la función de activación del cuerpo de la propia neurona. Se añade como un elemento posterior al propio perceptrón. Al eliminarse la función de activación del perceptrón su salida ya no es un valor binario, sino real. Ahora el hiperplano de separación no divide los puntos en aquellos que activan o no la salida, sino en puntos que generan un valor real positivo (los que están por encima del hiperplano) y los que generan un valor real negativo (los que caen por debajo del hiperplano). En el trabajo original, los autores aplican una función escalonada a la salida del perceptrón, aunque no se considera parte del per-

ceptrón; incluso se podría prescindir de ella dependiendo del problema o utilizar una función diferente.

Además del perceptrón, como se verá en la Sección 2.6, Rosenblatt propuso un mecanismo para ajustar el valor de sus pesos. A partir de este momento, ya no es necesario definir a mano un valor umbral diferente para cada problema: se puede establecer un valor fijo y ajustarlo según el problema.

Algunas de las mejoras que presenta el perceptrón con respecto a la neurona de McCulloch-Pitts fueron introducidas por ADALINE (de *adaptative linear element*), un modelo posterior al perceptrón, creado por el profesor Bernard Widrow y su alumno Ted Hoff en 1960 [4]. En concreto, ADALINE fue el primero en introducir el umbral como una entrada del perceptrón y separar la función de activación de este para que su salida consista únicamente en la combinación lineal de las entradas. Estas mejoras fueron añadidas al perceptrón original y es la versión que conocemos hoy día. En la Figura 1.6 se muestra el esquema del perceptrón.



**Figura 1.6:** El perceptrón

Se observa en la Figura 1.6 cómo ahora la agregación se produce sobre el resultado de multiplicar la entrada por su peso en lugar de utilizar únicamente la entrada. Además, la entrada pasa a ser un valor real y no binario. Por lo tanto, con este modelo es posible crear sistemas de decisión que trabajen sobre variables reales e incluso dar más importancia a algunas sobre otras mediante los pesos.

Es habitual describir a un perceptrón con su ecuación en notación vectorial (la función de activación es independiente del perceptrón) como se muestra en la Ecuación 1.1. Las entradas se consideran como un único vector al igual que los pesos:  $x = [x_0, \dots, x_n]^T$ ,  $w = [w_0, \dots, w_n]^T$ .

$$y = w^T x \quad (1.1)$$

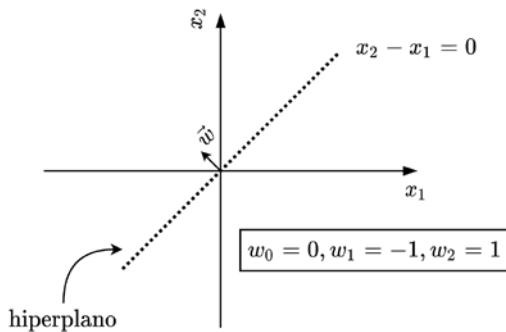
En la literatura se puede encontrar otra descripción del perceptrón donde la entrada correspondiente al umbral,  $\theta$ , se denota de una forma diferente. Como es sabido, esta entrada representa un sesgo para el sistema que se fija a un valor, normalmente a uno; por lo que muchos autores optan por denotar al peso que multiplica a esta entrada con la letra  $b$  de *bias* (sesgo en español). En este caso la ecuación del perceptrón queda como se muestra en la Ecuación 1.2. Al igual que antes, las entradas se consideran como un único vector al igual que los pesos:  $x = [x_1, \dots, x_n]^T$ ,  $w = [w_1, \dots, w_n]^T$ .

$$y = b + w^T x \quad (1.2)$$

Otras de las ecuaciones relacionadas con el perceptrón es la ecuación del hiperplano que viene viene definida por los pesos como se indica en la Ecuación 1.3.

$$w^T x = 0 \quad (1.3)$$

El hiperplano viene definido por el vector normal,  $w = [w_0, \dots, w_n]^T$ , que es perpendicular al hiperplano (ver Figura 1.7). El vector normal siempre apunta a la región del espacio donde quedan los puntos que hacen que la salida del perceptrón sea positiva, es decir, aquellos que satisfacen  $w^T x > 0$ .



**Figura 1.7:** El vector normal al hiperplano definido por los pesos del perceptrón

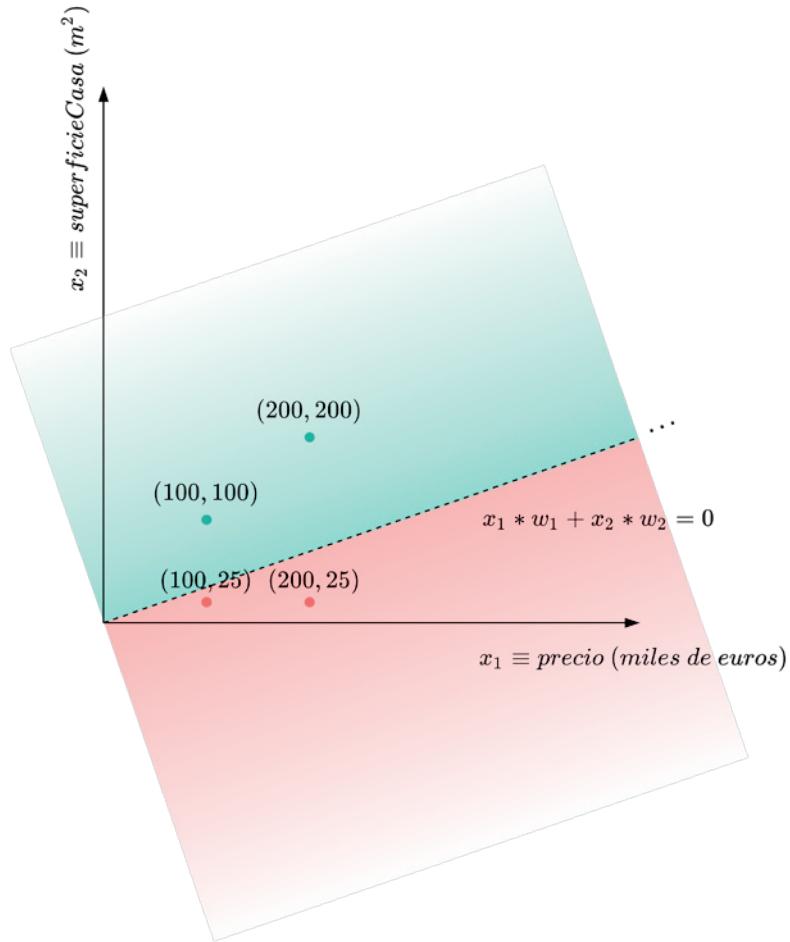
Con el siguiente ejemplo se puede ver algunas de las nuevas aplicaciones del perceptrón.

**Ejemplo 1.2 :** Sistema para decidir si comprar o no una casa en base a su precio (en miles de euros) y superficie (en m<sup>2</sup>).

El Ejemplo 1.2 no podría modelarse con la neurona de McCulloch-Pitts, ya que las entradas son variables reales. Además, como se estudiará en las siguientes secciones, se puede construir un conjunto de datos de ejemplo donde se indique qué salida del modelo se desea para ciertos valores de precio y superficie. Posteriormente, el propio modelo ajustará sus pesos para imitar el comportamiento inherente en los datos mediante un algoritmo de entrenamiento.

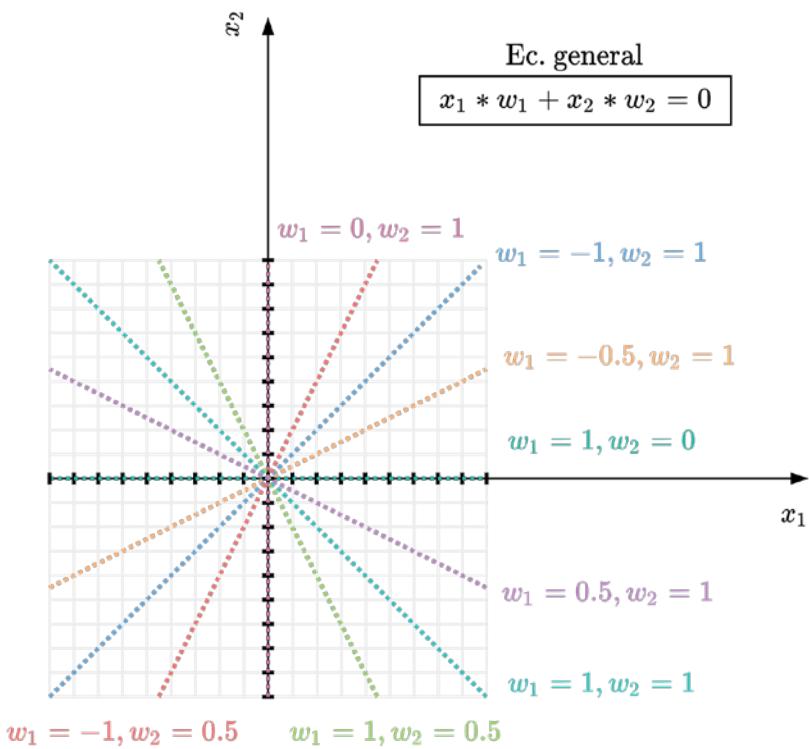
En la Figura 1.8 se muestra un perceptrón (sin *bias*) para el Ejemplo 1.2. Se observa que cualquier punto del espacio de características ( $\mathbb{R}^2$  al haber dos características: precio y tamaño) puede ser una entrada válida para el modelo. A pesar de que la salida es un valor real, se utiliza la función de activación escalonada (la función escalón unitario) para dar a la salida un valor binario que representa la activación de la neurona. Cuando se habla de puntos que activan la salida son aquellos que

generan un valor positivo a la salida del perceptrón, que serán los que generen el valor uno a la salida de la función de activación.



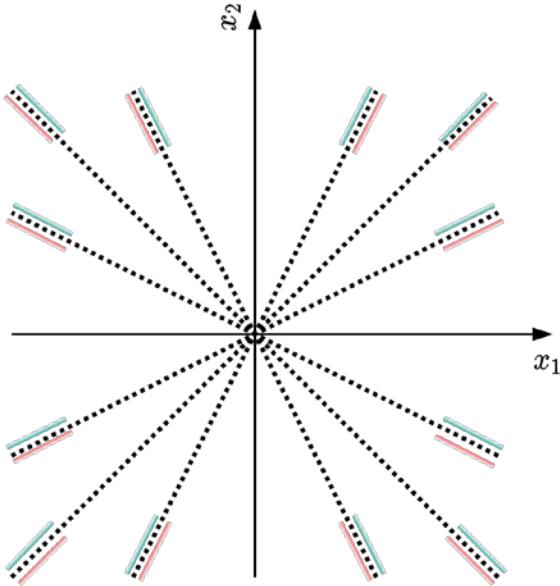
**Figura 1.8:** Perceptrón (sin *bias*) para decidir si comprar o no una casa. Se indican algunos puntos a modo de ejemplo, pero el modelo es capaz de clasificar cualquier punto del plano  $\mathbb{R}^2$ . El color verde indica los puntos que activan la salida del modelo y los rojos los que no.

Si se observa el hiperplano del perceptrón, se ve que el efecto que tiene utilizar los pesos es que permiten su rotación. En función del valor de  $w_1$  y  $w_2$  el hiperplano presentará más o menos pendiente, con la posibilidad de que sea tanto positiva como negativa. En la Figura 1.9 se muestran algunos hiperplanos de ejemplo en función del valor de  $w_1$  y  $w_2$  y en la Figura 1.10 se muestra la división del espacio que hace cada hiperplano, es decir, los puntos del espacio que activan la salida (en color verde) y los que no (en color rojo).



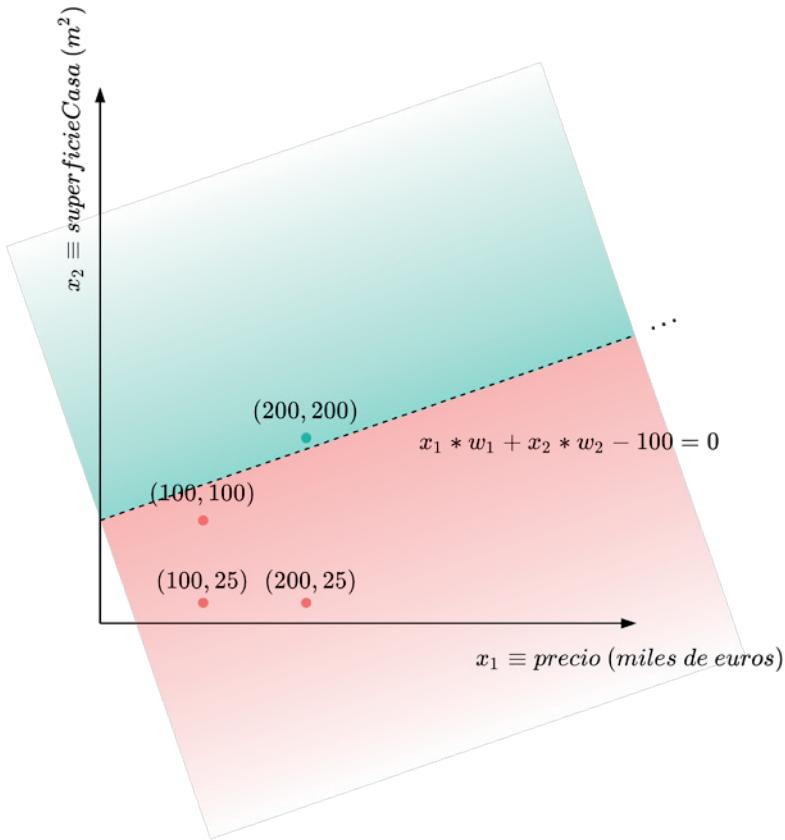
**Figura 1.9:** Ejemplos de hiperplanos en función del valor de  $w_1$  y  $w_2$  (con  $w_0 = 0$ )

Como el perceptrón cuenta con un término independiente para definir su hiperplano (gracias a utilizar el umbral como entrada del perceptrón), se podrían crear algunos modelos cuyos hiperplanos no tengan que pasar por el origen. Esto sería



**Figura 1.10:** División del espacio para diferentes hiperplanos de ejemplo. En color verde se indica la parte del espacio que genera un valor positivo a la salida de cada perceptrón y en rojo la que genera un valor negativo.

necesario si tuviéramos una restricción del tipo «Comprar la casa si tiene, como mínimo,  $100 \text{ m}^2$ ». En la Figura 1.11 se muestra el hiperplano del perceptrón del Ejemplo 1.2 si le añadiésemos esta restricción.



**Figura 1.11:** Perceptrón para decidir si comprar o no una casa si esta tiene, como mínimo,  $100 \text{ m}^2$ . Se indican algunos puntos a modo de ejemplo, pero el modelo es capaz de clasificar cualquier punto del plano  $\mathbb{R}^2$ . El color verde indica los valores que activan la salida del modelo y los rojos los que no.

**Tabla 1.1:** Salidas de una neurona de McCulloch-Pitts para el Ejemplo 1.1 con 3 entradas y  $\theta = 3$ 

$x_1$	$x_2$	$x_3$	$g(\mathbf{x}) = x_1 + x_2 + x_3$	$y = f(g(\mathbf{x})) = (x_1 + x_2 + x_3) \geq \theta$ ¿VoyAlCine?
estoyAlegre	generoAventuras	actuaJohnnyDepp		
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	2	0
1	0	0	1	0
1	0	1	2	0
1	1	0	2	0
1	1	1	3	1

### 1.2.1. Funciones de activación

Como se ha estudiado en la sección anterior, con la introducción del perceptrón, la función activación deja de formar parte de este y pasa a ser un elemento posterior al perceptrón, que actúa sobre su salida. Permite modificar la salida del perceptrón para que esta tenga sentido en base al problema que se esté abordando.

En los comienzos, debido a que su aplicación se limitaba a las funciones booleanas, la neurona de McCulloch-Pitts empleaba la función escalonada, con el valor uno para representar la activación y el cero para la no activación. Sin embargo, con la llegada del perceptrón se pasó a trabajar con valores reales, lo que ampliaba sus posibles aplicaciones. Si en el Ejemplo 1.2 se quisiera utilizar un perceptrón para que indique una probabilidad de comprar o no una casa, no se debería utilizar la función escalón, pero si no usamos ninguna función de activación la salida de la red será una combinación lineal de las entradas, que no está restringida al rango  $[0, 1]$ .

Es importante notar que las entradas podrían ser incluso negativas (aunque no tenga sentido un precio o tamaño negativo), ya que la frontera de decisión se extiende por todo el espacio, incluyendo valores negativos. El control de las entradas escapa del perceptrón y se debe comprobar que están en el rango de valores permitidos. No obstante, sí que se puede controlar el rango y tipo de los valores que queremos a la salida.

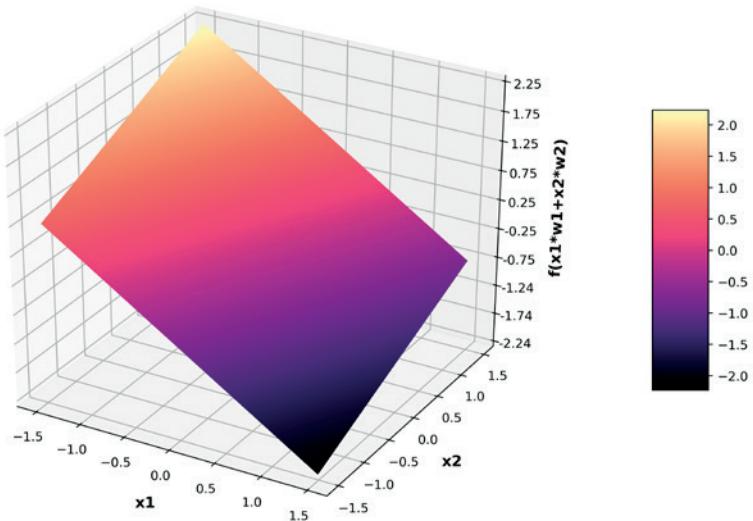
Para limitar el rango de valores de salida del percetrón hay que utilizar funciones diferentes a la función escalonada. Conforme aparecieron nuevas aplicaciones del perceptrón fueron necesarias funciones con distintos rangos de valores. A pesar de que se podría emplear cualquier función, en la Tabla 1.2 se recogen las funciones de activación más populares junto con algunas de sus propiedades.

No hay que olvidar que el efecto de la función de activación es la modificación del valor de salida, pero no del hiperplano del perceptrón. En las Figuras 1.12, 1.13, 1.14, 1.15 y 1.16 se muestran el efecto de las funciones de activación de la Tabla 1.2 en el valor de salida de un perceptrón de ejemplo con dos entradas ( $x_1$  y  $x_2$ ) y sin sesgo ( $w_0 = 0$ ).

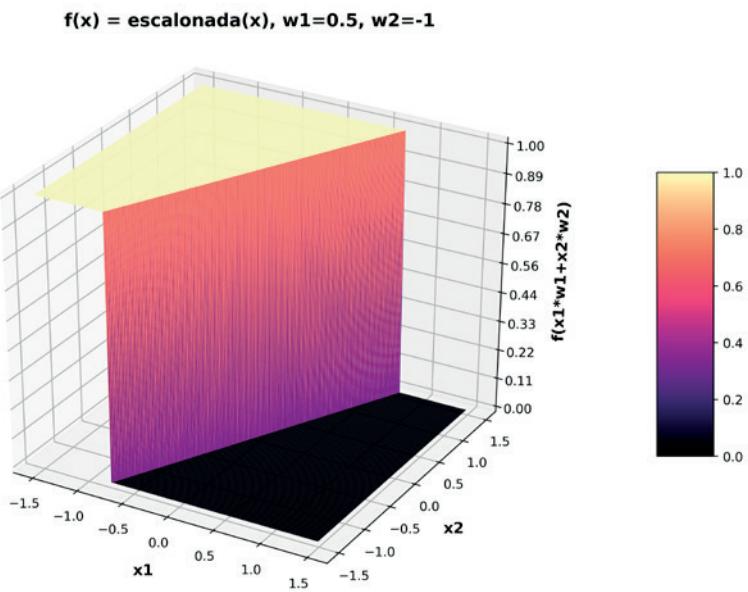
**Tabla 1.2:** Principales funciones de activación

Nombre	Gráfica	Ecuación	Rango
Identidad		$f(x) = x$	$(-\infty, \infty)$
Escalonada o umbral		$f(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$	$\{0, 1\}$
Logística o sigmoidal		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$(0, 1)$
Tangente hiperbólica (tanh)		$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$(-1, 1)$
Unidad lineal rectificada (ReLU por sus siglas en inglés)		$f(x) = \begin{cases} x & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$	$[0, \infty)$

$$f(x) = x, w_1=0.5, w_2=-1$$

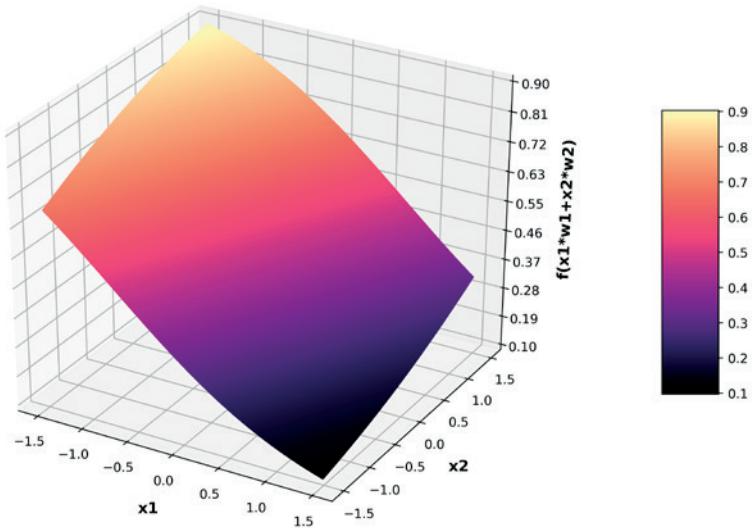


**Figura 1.12:** Perceptrón con dos entradas ( $x_1$  y  $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación identidad



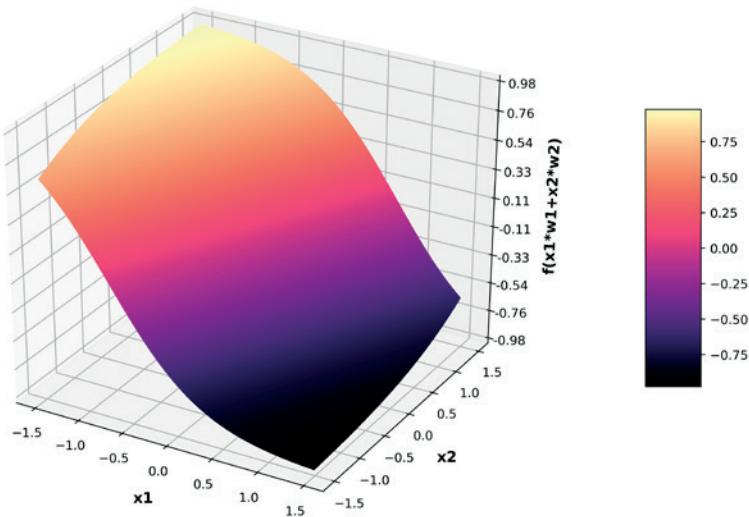
**Figura 1.13:** Perceptrón con dos entradas ( $x_1$  y  $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación escalonada

$$f(x) = \text{sigmoide}(x), w_1=0.5, w_2=-1$$



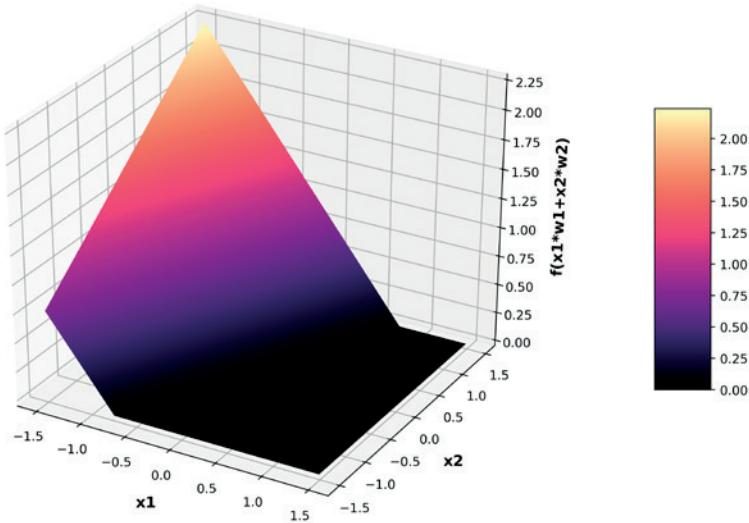
**Figura 1.14:** Perceptrón con dos entradas ( $x_1$  y  $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación sigmoide

$$f(x) = \tanh(x), w_1=0.5, w_2=-1$$



**Figura 1.15:** Perceptrón con dos entradas ( $x_1$  y  $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación tanh

$$f(x) = \text{ReLU}(x), w_1=0.5, w_2=-1$$



**Figura 1.16:** Perceptrón con dos entradas ( $x_1$  y  $x_2$ ), sin sesgo ( $w_0 = 0$ ) y con función de activación ReLU

## 1.2.2. Principales aplicaciones: clasificación y regresión

Existen multitud de problemas en los que se puede utilizar un perceptrón para resolverlos. Como bien se sabe, las técnicas de *machine learning* se caracterizan por utilizar un conjunto de datos para resolver los problemas. Uno de los problemas más habituales dentro del *machine learning* es el **modelado predictivo**. Con el modelado predictivo se pretende crear modelos que utilicen los datos disponibles para buscar patrones y tratar de predecir el resultado para datos nunca vistos mediante la formulación de hipótesis.

En el caso del perceptrón, la hipótesis se corresponde con la ecuación del hiperplano, ya que es la que define su salida para cualquier valor de entrada, lo que permite predecir la salida para datos nunca vistos. En el Ejemplo 1.2, el perceptrón puede dar una salida para cualquier dato de entrada que se desee, incluido aquellos que nunca haya estudiado (como casas de un millón de euros o de un millón de metros cuadrados). El perceptrón no tiene limitada su entrada  $y$ , aunque no haya trabajado con estos datos a la hora de definir sus pesos, puede dar (*predecir*) una salida para estos valores.

El modelado predictivo puede ser visto como un problema matemático de **aproximación de funciones**. Se desea obtener la transformación que hay que aplicar a los datos de entrada para conseguir la salida esperada. Es decir, dada una entrada,  $x \in \mathbb{R}^n$ ; y un valor de salida esperado,  $y \in \mathbb{R}$ ; se busca obtener la transformación

$$f : \mathbb{R}^n \rightarrow \mathbb{R}$$

que hay que aplicarle a  $x$  para obtener  $y$ .

Los problemas que son abordados con el perceptrón (y con redes neuronales) son descritos en términos de aproximación de funciones. El objetivo será definir los pesos de forma que la ecuación del hiperplano,  $f'$ , sea lo más parecida posible,  $|f' - f| < \epsilon$ , a una función objetivo,  $f$ . En el Ejemplo 1.2, se busca aproximar la

función que, dado el precio y la superficie de una casa, devuelva un valor binario que indica si se compra la casa o no, es decir,

$$\begin{aligned} f : \quad & \mathbb{R}^2 \quad \rightarrow \quad \{0, 1\} \\ & (\text{precio}, \text{superficieCasa}) \quad \mapsto \quad \text{:comprar?} \end{aligned}$$

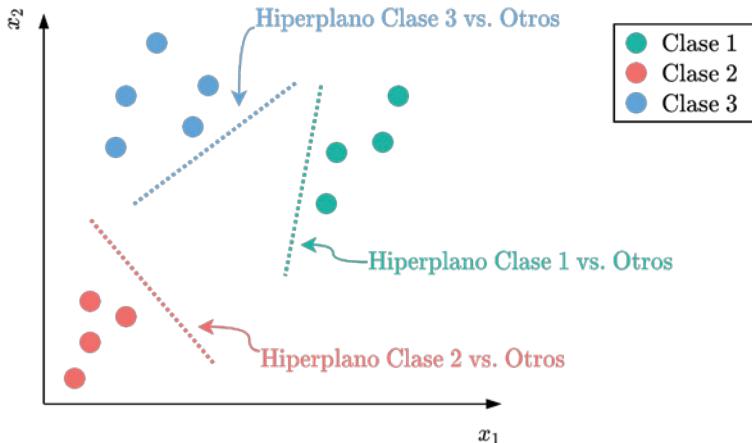
Dentro del modelado predictivo se distinguen dos categorías principales, clasificación y regresión:

- **Clasificación.** En esta categoría entra el Ejemplo 1.2. Los problemas de clasificación consisten en aproximar una función que transforme los datos de entrada a una salida de valores discretos (cero y uno en el Ejemplo 1.2). Dependiendo del número de valores de salida se distingue la clasificación binaria, para dos valores; y la clasificación multiclase para más de dos valores de salida. A continuación, se muestra un ejemplo de clasificación binaria y de clasificación multiclase.

- *Detectar tumores malignos.* Dada la edad del paciente y el tamaño del tumor se puede construir un perceptrón que realice una clasificación binaria para decidir si el tumor es maligno o benigno.
- *Detectar el número que hay dibujado en una imagen.* Se trata de un problema de clasificación multiclase. Una imagen puede ser representada como un vector de números. Cada píxel de la imagen es una entrada del perceptrón. La clasificación multiclase puede ser abordada desde dos perspectivas diferentes. Por un lado, se pueden utilizar diez perceptrones para que cada uno realice una clasificación binaria de cada dígito del cero al nueve; la clasificación final se podría hacer tomando la clase del perceptrón que genera un valor más alto. Por otro lado, como se estudia en la Sección 1.3, se podría utilizar una red neuronal con diez neuronas de forma que la salida de cada neurona represente la probabilidad de pertenencia de la entrada a cada una de las clases.

La primera opción se conoce como una clasificación uno contra todos y la segunda como clasificación multiclase, que se suelen abordar con redes neuronales. En la Figura 1.17 se muestra un ejemplo de clasificación multiclase (tres clases) con el enfoque uno contra todos (tres perceptrones, uno por ca-

da clase, para discernir entre los datos que son de la propia clase y los que no). Se observa que cada hiperplano separa los datos de su clase del resto.



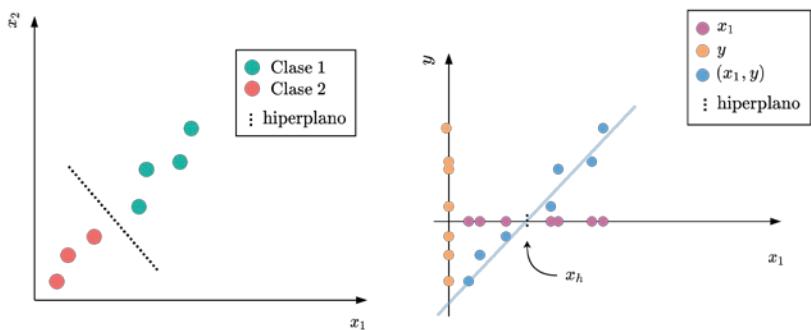
**Figura 1.17:** Clasificación multiclas con el enfoque uno contra todos

- **Regresión.** Los problemas de regresión consisten en aproximar una función para transformar los datos de entrada en un valor continuo de salida. Para este tipo de problemas no se pueden utilizar funciones de activación con una salida discreta como la función umbral. A continuación, se dan algunos ejemplos de problemas de regresión.
  - *Estimar el precio de una vivienda en base a su superficie.* Este problema es parecido al Ejemplo 1.2. El problema de regresión consiste en la estimación, en miles de euros, del precio de la vivienda. Utilizando la función de activación identidad es posible conseguir una salida continua.
  - *Dar la altura de una persona presente en una imagen.* Este problema se puede abordar utilizando la función de activación ReLU a la salida del perceptrón, ya que el valor de salida está en el rango  $[0, \infty)$  y permite descartar valores de altura negativos.

Tanto la clasificación como la regresión pueden recibir valores discretos o continuo de entrada, en lo único que se diferencian es en el tipo de dato de la salida. En el

caso de la clasificación, el tipo de dato de salida es discreto: se trabaja con una etiqueta para cada clase del problema, ya sea binaria (p. ej. positivo vs. negativo) o  $n$ -aria (p. ej. rubio vs. moreno vs. pelirrojo). En el caso de la regresión, la salida es continua: predice una cantidad (altura, precio, etc.).

En la Figura 1.18 se muestra un ejemplo de cada tipo de problema. Se observa cómo en el caso de la clasificación, en  $\mathbb{R}^2$ , el hiperplano del perceptrón trata de situarse en la zona intermedia entre las dos clases para que una de ellas provoque la activación del perceptrón y la otra no. En el caso de la regresión, en  $\mathbb{R}$ , el hiperplano (un punto) se sitúa también en la zona intermedia de los datos de entrada de forma que la distancia de los puntos al hiperplano es el valor que se utilizará para obtener la salida (en el eje  $y$ ).



**Figura 1.18:** Hiperplano (línea punteada) en un problema de clasificación (izquierda) y regresión (derecha)

Los problemas de clasificación pueden ser abordados como problemas de regresión y viceversa como se muestra a continuación.

- **Clasificación como regresión.** Si hay que clasificar una imagen en base a que contenga o no una persona, en lugar de utilizar una función de activación umbral, se puede emplear una función sigmoide e interpretar su salida como la probabilidad de que una persona esté o no en la imagen. Si se aplica un umbral a esta probabilidad se podrá hacer la clasificación de la entrada. El perceptrón

trata el problema como una regresión del valor de probabilidad esperado para cada entrada.

- **Regresión como clasificación.** Los problemas de regresión también pueden predecir valores discretos o ser asignados a una clase, aunque no son realmente etiquetas de clases, sino que se discretiza el valor de salida del perceptrón. La discretización puede consistir en redondear el valor de salida para obtener cantidades enteras o dividir el rango de valores de salida en las clases de interés. Por ejemplo, si se mide la altura en metros de una persona en una imagen, se pueden definir tres clases: baja,  $y \in [0, 1.6]$ ; normal,  $y \in (1.6, 1.85]$ ; y alta,  $y \in (1.85, \infty)$ . Cuando se obtiene el valor continuo de salida se comprueba en qué rango está para dar la etiqueta de la clase correspondiente como salida.

### 1.2.3. Limitaciones

Cuando se dio a conocer el perceptrón se crearon grandes expectativas en cuanto a lo que se podría hacer con este modelo. Prueba de ello es el siguiente comunicado de prensa del New York Times cuando se publicó el perceptrón en el año 1958.

*The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence. [...] Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech and writing in another language.*

*New Navy Device Learns By Doing. (7 de julio de 1958).*

*New York Times, p.25*

En el año 1969, Marvin Minsky y Seymour Papert publicaron el libro *Perceptrons*. En esta obra, Minsky y Papert hacen un riguroso estudio del perceptrón y sus limitaciones. Su principal conclusión es que el perceptrón solo puede resolver problemas que sean linealmente separables.

Un problema se considera que es linealmente separable si se puede trazar un hipérplano que divida el espacio en dos partes de forma que todos los puntos de

una clase caigan en una de las partes y los de la otra clase en la otra parte (ver Definición 1.1).

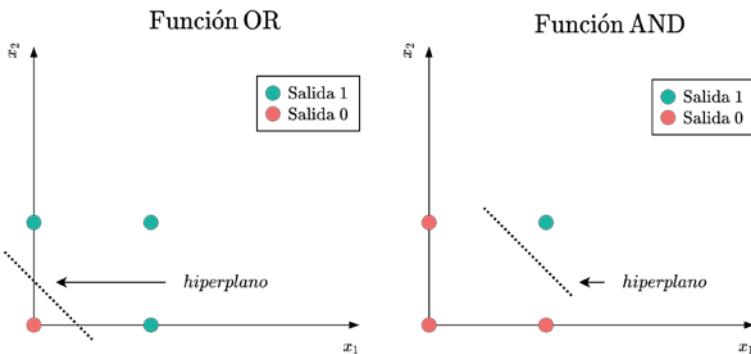
**Definición 1.1 :** Dos conjuntos de puntos,  $\mathcal{X}_1$  y  $\mathcal{X}_2$ , de un espacio euclídeo de dimensión  $n$ , se dice que son linealmente separables si

$$\exists w_1, \dots, w_n, \theta \in \mathbb{R} \mid \forall x \in \mathcal{X}_1 : \sum_{i=1}^n x_i * w_i > \theta \wedge \forall x \in \mathcal{X}_2 : \sum_{i=1}^n x_i * w_i < \theta$$

donde  $x_i$  es la componente  $i$ -ésima de  $x$ . En notación vectorial

$$\exists w \in \mathbb{R}^N, \theta \in \mathbb{R} \mid \forall x \in \mathcal{X}_1 : w^T x > \theta \wedge \forall x \in \mathcal{X}_2 : w^T x < \theta$$

Ejemplos de problemas separables linealmente son la función OR y la función AND, ya que se puede definir un hiperplano que separa los valores que activan la salida de los que no (ver figura 1.19).



**Figura 1.19:** Ejemplo de separación lineal mediante un hiperplano para la función OR y AND

Minsky y Papert demostraron que, si se dispone de un conjunto de datos finito y linealmente separable, se pueden hallar los pesos del hiperplano que separa correctamente los datos mediante un algoritmo de entrenamiento, que se estudiará en la Sección 2.6.

No obstante, existen muchas funciones que no son separables linealmente. En el caso de funciones booleanas, si se dispone de  $n$  variables, se pueden construir hasta  $2^{2^n}$  funciones diferentes (ver Tabla 1.3), pero, como se muestra en la Tabla 1.4, con el aumento del número de variables disminuye drásticamente el ratio de funciones que son separables linealmente respecto del total. Por lo tanto, se reduce el número de casos en los que se pueden utilizar el perceptrón.

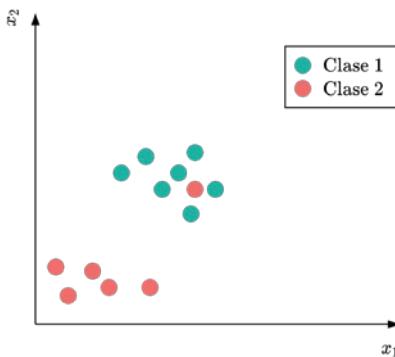
**Tabla 1.3:** Funciones booleanas diferentes para dos variables

$x_1$	$x_2$	$f_1$	$f_2$	$f_3$	$f_4$	$f_5$	$f_6$	$f_7$	$f_8$	$f_9$	$f_{10}$	$f_{11}$	$f_{12}$	$f_{13}$	$f_{14}$	$f_{15}$	$f_{16}$
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
0	1	0	0	0	0	1	1	1	0	0	0	0	1	1	1	1	
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	

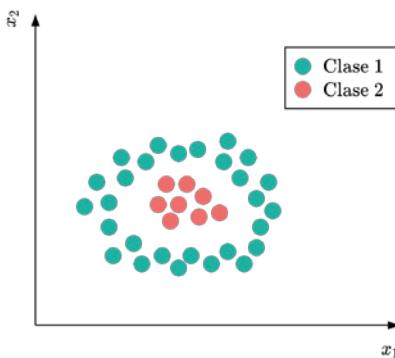
**Tabla 1.4:** Funciones booleanas diferentes, separables linealmente y ratio en función del número de variables

Número de variables	Funciones booleanas	Funciones separables	Ratio
2	16	14	8.7500 e-01
3	256	104	4.0625 e-01
4	65536	1882	2.8717 e-02
5	4.2949 e+09	9.4572 e+04	2.2019 e-05
6	1.8446 e+19	1.5028 e+07	8.1467 e-13
7	3.4028 e+38	8.3780 e+09	2.4620 e-29

Además, la mayoría de los problemas reales no son separables linealmente. En algunas situaciones debido a la existencia de *outliers* en los datos (ver Figura 1.20), en otras es inherente a los propios datos (ver Figura 1.21). Este último es el caso de la función XOR (ver Figura 1.22), que fue estudiada por Minsky y Papert en su libro donde demostraron que el perceptrón es incapaz de aproximar esta función.

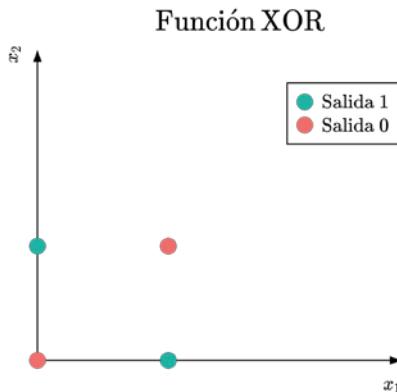


**Figura 1.20:** Ejemplo de *outlier* en los datos



**Figura 1.21:** Ejemplo de datos no separables linealmente

Aun así, se podría resolver el problema XOR si se emplea más de un perceptrón, mediante una red de perceptrones, lo que se conoce hoy día como **red neuronal**. Una posible solución sería utilizar tres perceptrones. Uno de ellos trazaría un hipoplano por encima del punto situado en el origen y se activaría con todos los puntos excepto con el  $(0, 0)$ . Este perceptrón no es más que la función OR (ver Figura 1.23). Otro perceptrón haría lo mismo pero al revés, es decir, su salida se activaría si entra cualquier punto que no sea el  $(1, 1)$ . Un perceptrón con la fun-

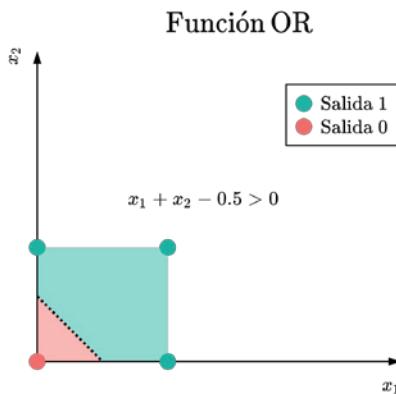


**Figura 1.22:** Función XOR

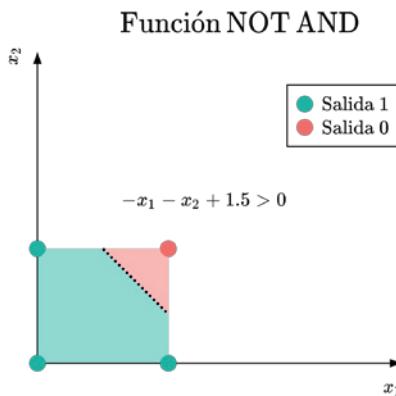
ción NOT AND conseguiría este comportamiento (ver Figura 1.24). Por último, con un tercer perceptrón que utilice la función AND (ver Figura 1.25) se obtendría una salida activa si las dos entradas están activas. Este comportamiento de la función AND se puede aprovechar para, en lugar de recibir como entrada  $x_1$  y  $x_2$ , recibir la salida de los dos perceptrones anteriores. Con este pequeño truco se lograría aproximar la función XOR (ver Figura 1.26).

Se ha podido representar la función XOR gracias a crear una composición de funciones con los tres perceptrones. El primero realiza la función OR de las entradas,  $x_1$  OR  $x_2$ ; el segundo la función NOT AND de las entradas, NOT ( $x_1$  AND  $x_2$ ); y el tercero la función AND sobre la salida de los dos perceptrones anteriores, ( $x_1$  OR  $x_2$ ) AND (NOT ( $x_1$  AND  $x_2$ )).

Para ver qué transformación sufren los datos, se pueden representar las salidas de cada uno de los tres perceptrones (ver Figuras 1.27, 1.28 y 1.29). Hay que tener presente que el tercer preceptrón recibe como entrada la salida de los otros dos y no los dos valores de entrada originales,  $x_1$  y  $x_2$ . Se observa cómo el último de los perceptrones solo deja activa la parte en la que los dos anteriores están activos. Por último, en la Tabla 1.5 se calculan todas las salidas para cada uno de los perceptrones, lo que permite comprobar que la red de tres perceptrones ha aproximado correctamente a la función XOR.



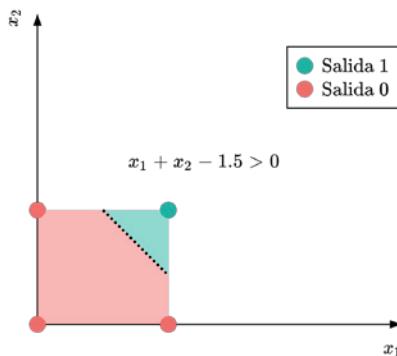
**Figura 1.23:** Salida de perceptrón que aproxima la función OR



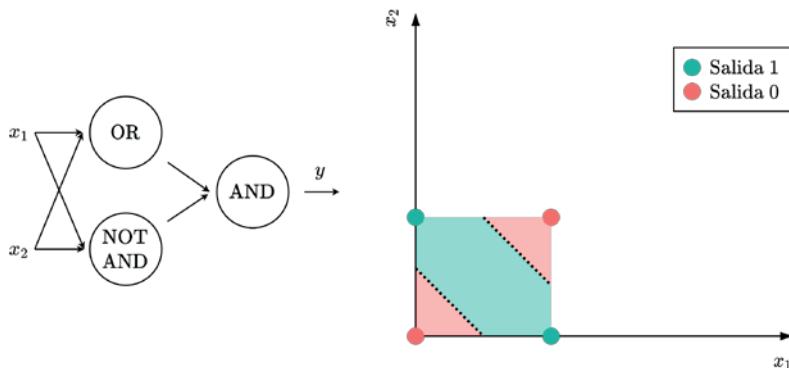
**Figura 1.24:** Salida de perceptrón que aproxima la función NOT AND

Existe cierta polémica en torno a si Minsky y Papert conocían que el problema XOR se podía solucionar con una red de perceptrones o no. Aun así, la mayoría de los investigadores extrapolaron las limitaciones del perceptrón a las redes neuronales que, a pesar de no ser ciertas, tampoco se conocía un algoritmo que permitiese ajustar los pesos de estas redes. Todo lo anterior unido a la aparición de nuevos

Función AND

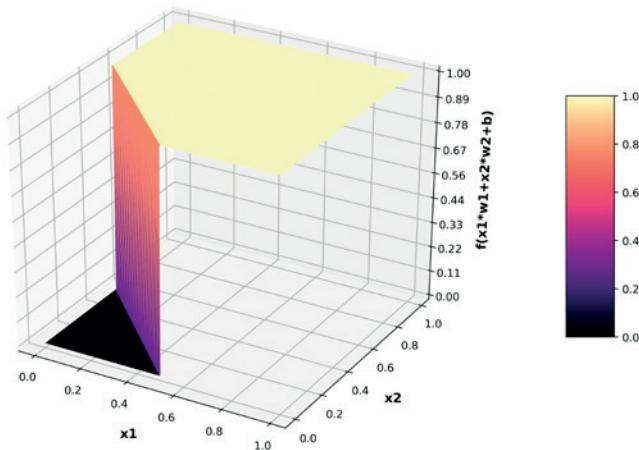
**Figura 1.25:** Salida de perceptrón que aproxima la función AND

Función XOR

**Figura 1.26:** Tres perceptrones逼近 la función XOR

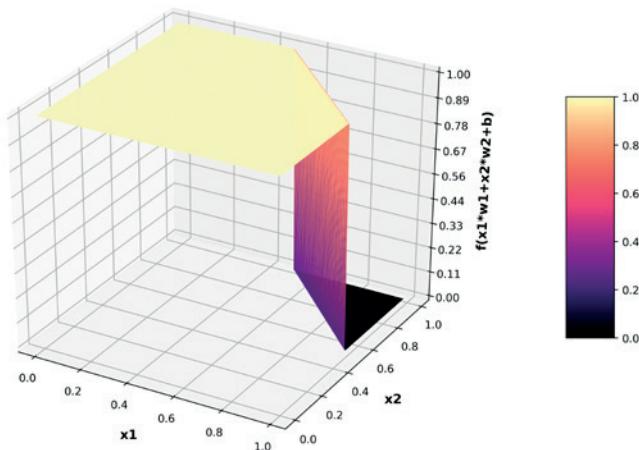
algoritmos más prometedores basados en el simbolismo propiciaron el abandono de las redes neuronales y afectó en gran medida a la financiación de proyectos relacionados con el conexionismo durante gran parte de los años 80. Es lo que se conoce como el primer invierno de la inteligencia artificial.

$$f(x) = \text{umbral}(x), w1=1, w2=1, b=-0.5$$



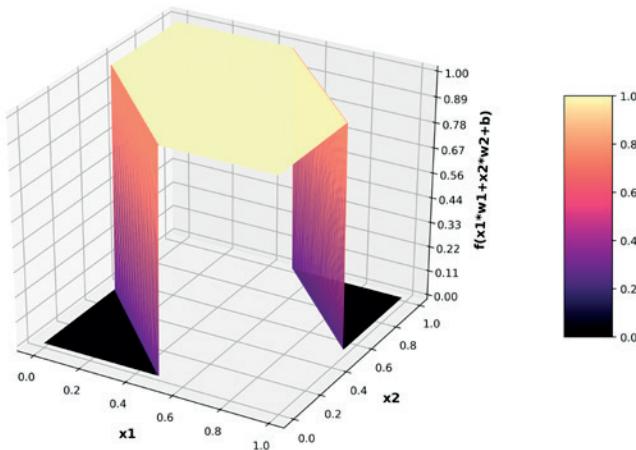
**Figura 1.27:** Salida de un perceptrón que aproxima la función OR

$$f(x) = \text{umbral}(x), w1=-1, w2=-1, b=1.5$$



**Figura 1.28:** Salida de un perceptrón que aproxima la función NOT AND

$$f(x) = \text{umbral}(x, w_1=1, w_2=1, b=-1.5)$$



**Figura 1.29:** Salida construida con tres perceptrones para aproximar la función XOR. Los pesos indicados son para el último perceprón que implementa la función AND sobre la salida de los otros dos.

**Tabla 1.5:** Entradas y salidas de cada uno de los perceptrones de la red neuronal para aproximar la función XOR

$x_1$	$x_2$	Función OR $y_1 = x_1 + x_2 - 0.5 > 0$	Función NOT AND $y_2 = -x_1 - x_2 + 1.5 > 0$	Función XOR $(x_1 \text{ } OR \text{ } x_2) \text{ } AND \text{ } (NOT(x_1 \text{ } AND \text{ } x_2))$ $y_1 + y_2 - 1.5 > 0$ $(x_1 \text{ } OR \text{ } x_2) + (NOT(x_1 \text{ } AND \text{ } x_2)) - 1.5 > 0$
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

## 1.3. Redes neuronales

Como se vio en la sección anterior, el perceptrón es lo que se conoce hoy día como neurona artificial. La combinación de varios perceptrones forma un perceptrón multicapa, también conocido como red neuronal artificial o, simplemente, red neuronal. En esta sección se estudia en profundidad estas redes, sus aplicaciones y limitaciones.

### 1.3.1. Arquitectura

Para describir una red neuronal se suele hacer referencia a su arquitectura. Esto es, la organización y conexiones de las neuronas de la red.

El primer componente propio de una red neuronal es la **capa**. Las neuronas que reciben las mismas entradas son agrupadas en la misma capa (ver Figura 1.30).

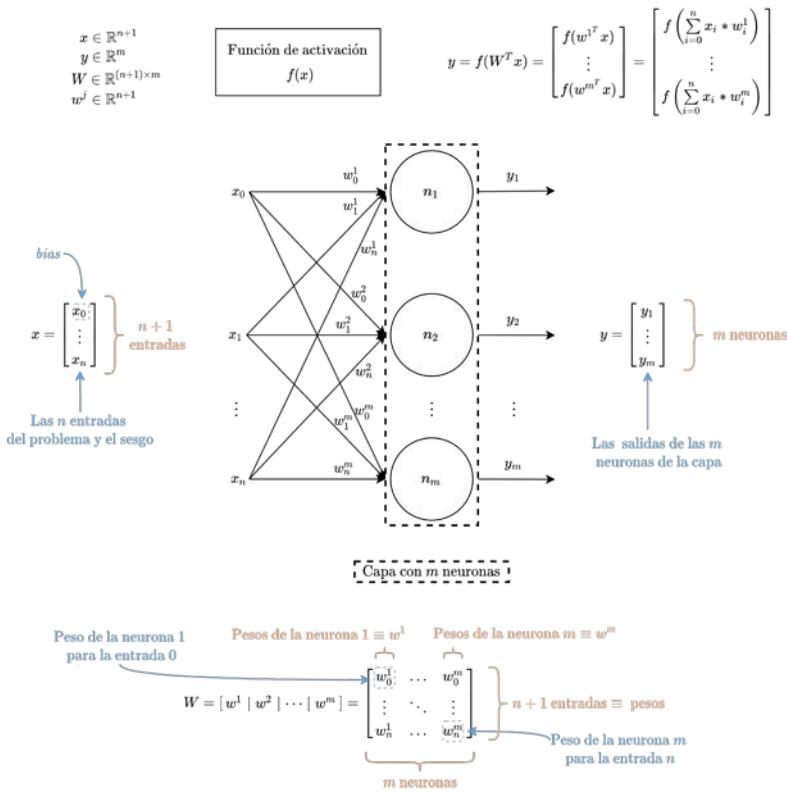
En las redes neuronales es común denotar las operaciones en formato vectorial para facilitar la lectura. Ya se conoce la ecuación de un perceptrón (neurona) en formato vectorial:  $y = w^T x$ . Por lo general, en una capa hay más de una neurona. Para mantener la notación vectorial, como se muestra en la Figura 1.30, las  $n$  entradas se agrupan en un único vector,  $x = [x_0, \dots, x_n]^T$ , en el que el primer componente,  $x_0$ , es el *bias*, normalmente  $x_0 = 1$ . Las salidas de las  $m$  neuronas de la capa son agrupadas también en un único vector,  $y = [y_1, \dots, y_m]^T$ , conocido como el vector de salida de la capa, con tantos componentes como neuronas haya en la capa. Por último, los pesos de cada neurona,  $w^i = [w_0^i, \dots, w_n^i]^T$ , son agrupados en la matriz de pesos,  $W$ , donde cada vector es una columna de la matriz. Si hay  $n + 1$  entradas ( $n$  entradas y el sesgo) y  $m$  neuronas en la capa, la matriz de pesos tendrá  $n + 1$  filas y  $m$  columnas,  $W \in \mathbb{R}^{(n+1) \times m}$ . Con esta nueva notación, el vector de salida de la capa se puede calcular como se muestra en la Ecuación 1.4.

$$y = W^T x \quad (1.4)$$

En la Ecuación 1.4 no se ha incluido la función de activación. Normalmente, las neuronas de una misma capa cuentan con la misma estructura, es decir, las mismas conexiones y funciones de activación y, dado que la función de activación se aplica

sobre el valor de salida de cada neurona por separado, se puede utilizar la notación de la Ecuación 1.5 para indicar que la función de activación,  $f$ , se aplica a cada componente del vector  $W^T x$ .

$$y = f(W^T x) \quad (1.5)$$



**Figura 1.30:** Capa de una red neuronal

Como se observa en la Figura 1.30, el término *bias*,  $x_0$ , es una entrada común para todas las neuronas, aunque cada una cuenta con un peso diferente para sus entradas.

Las capas que tienen todas las neuronas conectadas a todas las entradas se las conoce como capas completamente conectadas o *fully connected* en inglés.

Con una única capa ya se tiene una red neuronal. Una red puede tener cualquier número de capas. Cuando hay más de una capa se suelen nombrar de forma diferente según la posición que ocupan en la red (ver Figura 1.31):

- **Capa de entrada.** Es habitual que los datos de entrada (el vector  $x$ ) se represente como una capa más de la red (capa 0). En este caso, los elementos de la capa son los componentes del vector de entrada  $x$ , es decir,  $[x_0, \dots, x_n]^T$ , en lugar de neuronas, por lo que en algunos libros se suelen denotar a las «neuronas» de esta capa con la letra  $x$  en lugar de  $n$ . La salida de esta capa,  $y^0$ , será directamente el vector de entrada  $x$ . Esta capa solo se usa porque simplifica la notación de las operaciones.
- **Capa oculta y capa de salida.** Son las capas que están después de la capa de entrada. Sus neuronas son los perceptrones que se han estudiado en la Sección 1.2. Esto es, cuentan con pesos que se encargan de transformar los datos de entrada y con una función de activación. La única diferencia entre una capa oculta y la capa de salida es el lugar que ocupan en la red: la capa de salida es la última capa de la red y la capa oculta está entre la capa de entrada y la capa de salida.

Es importante notar que todas las capas, excepto la capa de salida, tienen el término *bias* representado como la neurona 0 de la capa, aunque no se suele dibujar junto con las demás al ser una «neurona» especial. Normalmente, se utiliza el mismo valor para todas las capas de la red y suele ser 1. Esta forma de utilizar el *bias* implica que, para todas las capas distintas de la capa 0, su entrada consistirá en la salida de la capa anterior y el valor de *bias*. Es decir, para una capa  $r$ , su entrada,  $x^r$ , consistirá en el vector resultado de concatenar la salida de la capa anterior,  $y^{r-1}$ , con el *bias* de esa capa,  $n_0^{r-1}$ , obteniéndose la entrada de la capa  $r$  como  $x^r = [n_0^{r-1} | y^{r-1}]^T$ .

La capa de entrada no se suele tener en cuenta a la hora de indicar el número de capas de una red, ya que no es una capa como las otras. La red más pequeña que se puede crear consistiría en una única capa (dos capas realmente, la de entrada y la de salida). Siempre existe una capa de salida, solo cuando hay una capa intermedia se puede hablar de capas ocultas.

Como se observa en la Figura 1.31, no todas las capas tienen el mismo número

de neuronas y, por tanto, salidas. Cuando se trabaja con una red neuronal se suele estar interesado en la salida de la red, la salida de la última capa; y no en las salidas de las capas intermedias. La salida de una red se calcula capa a capa, como una composición de funciones. La primera capa con perceptrones (oculta o de salida, dependiendo del número de capas) toma los valores de la capa de entrada y les aplica una transformación mediante la Ecuación 1.5. La salida de esta capa será la entrada de la siguiente, a la que se le vuelve a aplicar la Ecuación 1.5 (cada capa utiliza sus propios pesos). Cuando se llega a la última capa se obtiene la salida de la red. Se puede expresar este proceso mediante composición de funciones según se muestra en la Ecuación 1.6.

$$\begin{aligned} y^l &= f^l(W^{l^T} x^l) = f^l\left(W^{l^T} f^{l-1}\left(W^{l-1^T} x^{l-1}\right)\right) = \\ &= \dots = f^l\left(W^{l^T} f^{l-1}\left(\dots f^1\left(W^{1^T} x^1\right) \dots\right)\right) \end{aligned} \quad (1.6)$$

donde

- $y^l$  es la salida de la red. Obtenida calculando las salidas desde la primera hasta la última capa,  $l$ .
- $f^i$  es la función de activación de la capa  $i$ -ésima.
- $W^i$  es la matriz de pesos para la capa  $i$ .
- $x^i$  es la entrada de la capa  $i$ . Es el vector,  $x^i = [n_0^{i-1} | y^{i-1}]^T$ , resultado de concatenar la salida,  $y^{i-1}$ , y el *bias*,  $n_0^{i-1}$ , de la capa anterior,  $i - 1$ .

En el Ejemplo 1.3, se detallan los cálculos de la salida de una red neuronal que se utiliza para aproximar la función XOR.

**Ejemplo 1.3 :** Función XOR con una red neuronal de dos capas y tres neuronas como la presentada en la Figura 1.26. Los pesos de cada neurona y una arquitectura detalla de la red se muestran en la Figura 1.32. Según los pesos de las neuronas, la salida de la red se calcula como se muestra a continuación:

$$y = f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ f \left( \begin{bmatrix} -0.5 & 1.5 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \right) \end{bmatrix} \right)$$

donde  $f$  es la función umbral en 0.

Los datos de entrada y las salidas esperadas son las siguientes:

$x_1$	$x_2$	$y$
0	0	0
0	1	1
1	0	1
1	1	0

Aplicando la ecuación anterior a todas las entradas se observa que la red logra aproximar la función XOR correctamente:

- $x_1 = 0, x_2 = 0$ :

$$\begin{aligned} y &= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ f \left( \begin{bmatrix} -0.5 & 1.5 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) \end{bmatrix} \right) = \\ &= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ f \left( \begin{bmatrix} -0.5 \\ 1.5 \end{bmatrix} \right) \end{bmatrix} \right) = \end{aligned}$$

$$= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right) = \\ = f([-0.5]) = [0] = 0$$

■  $x_1 = 0, x_2 = 1$ :

$$y = f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \left[ \frac{1}{f \left( \begin{bmatrix} -0.5 & 1.5 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right)} \right] \right) = \\ = f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \left[ \frac{1}{f \left( \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \right)} \right] \right) = \\ = f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) = \\ = f([0.5]) = [1] = 1$$

■  $x_1 = 1, x_2 = 0$ :

$$\begin{aligned}
y &= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \left[ \frac{1}{f \left( \begin{bmatrix} -0.5 & 1.5 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \right)} \right] \right) = \\
&= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \left[ \frac{1}{f \left( \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \right)} \right] \right) = \\
&= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) = \\
&= f([0.5]) = [1] = 1
\end{aligned}$$

■  $x_1 = 1, x_2 = 1$ :

$$y = f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \left[ \frac{1}{f \left( \begin{bmatrix} -0.5 & 1.5 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right)} \right] \right) =$$

$$\begin{aligned}
&= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ f \left( \begin{bmatrix} 1.5 \\ -0.5 \end{bmatrix} \right) \end{bmatrix} \right) = \\
&= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \right) = \\
&= f([-0.5]) = [0] = 0
\end{aligned}$$

Una de las ventajas que se tiene al calcular las salidas de las capas de la red en notación vectorial es que permite generalizar el cálculo de la salida de la red a cualquier número de entradas. El vector de entrada,  $x$ , pasa a ser una matriz donde cada columna de la matriz es el vector de una de las entradas. Si se dispone de  $n$  vectores de entrada,  $\{x^1, \dots, x^n\}$ , donde  $x^i = [x_1^i, \dots, x_m^i]^T$ , la matriz de entrada de la red,  $X$ , es

$$X = [x^1 | \dots | x^n] = \begin{bmatrix} x_0 & \dots & x_0 \\ \vdash & & \vdash \\ x_1^1 & \dots & x_1^n \\ \vdots & \ddots & \vdots \\ x_m^1 & \dots & x_m^n \end{bmatrix}$$

Hay que tener en cuenta que, a cada columna de la matriz (vector de entrada), se le concatena el *bias*,  $x_0$ , por lo que  $X \in \mathbb{R}^{(m+1) \times n}$ .

Ahora, la salida de la red es una matriz,  $Y$ , en la que cada columna,  $y^i$ , es el vector de salida de la red para el vector de entrada,  $x^i$ . Si la red cuenta con  $k$  neuronas en la capa de salida y hay  $n$  vectores de entrada, la salida de la red es

$$Y = [y^1 | \dots | y^n] = \begin{bmatrix} y_1^1 & \dots & y_1^n \\ \vdots & \ddots & \vdots \\ y_k^1 & \dots & y_k^n \end{bmatrix}$$

A continuación, se muestra el cálculo de la salida de la red del Ejemplo 1.3 de forma simultánea para todas las entradas. Dada la matriz de entrada

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

El cálculo de la matriz de salida o salida de la red será

$$\begin{aligned} y &= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \left[ \frac{\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}}{f \left( \begin{bmatrix} -0.5 & 1.5 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}^T * \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}} \right)} \right] \right) = \\ &= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \left[ \frac{\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}}{f \left( \begin{bmatrix} -0.5 & 0.5 & 0.5 & 1.5 \\ 1.5 & 0.5 & 0.5 & -0.5 \end{bmatrix} \right)} \right] \right) = \end{aligned}$$

$$\begin{aligned}
 &= f \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}^T * \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \right) = \\
 &= f \left( \begin{bmatrix} -0.5 \\ 0.5 \\ 0.5 \\ -0.5 \end{bmatrix}^T \right) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}^T
 \end{aligned}$$

Dado que solo hay una neurona en la capa de salida, para una única entrada, la salida será un escalar. Al introducir más de una entrada (cuatro en este caso), la salida pasa a ser un vector con cuatro componentes, uno por cada entrada. Si la capa de salida tuviese más de una neurona, la salida, para una única entrada, sería un vector, en lugar de un escalar. Por lo que, al introducir más de una entrada, la salida será una matriz, con tantas columnas como entradas y filas como neuronas haya en la capa de salida.

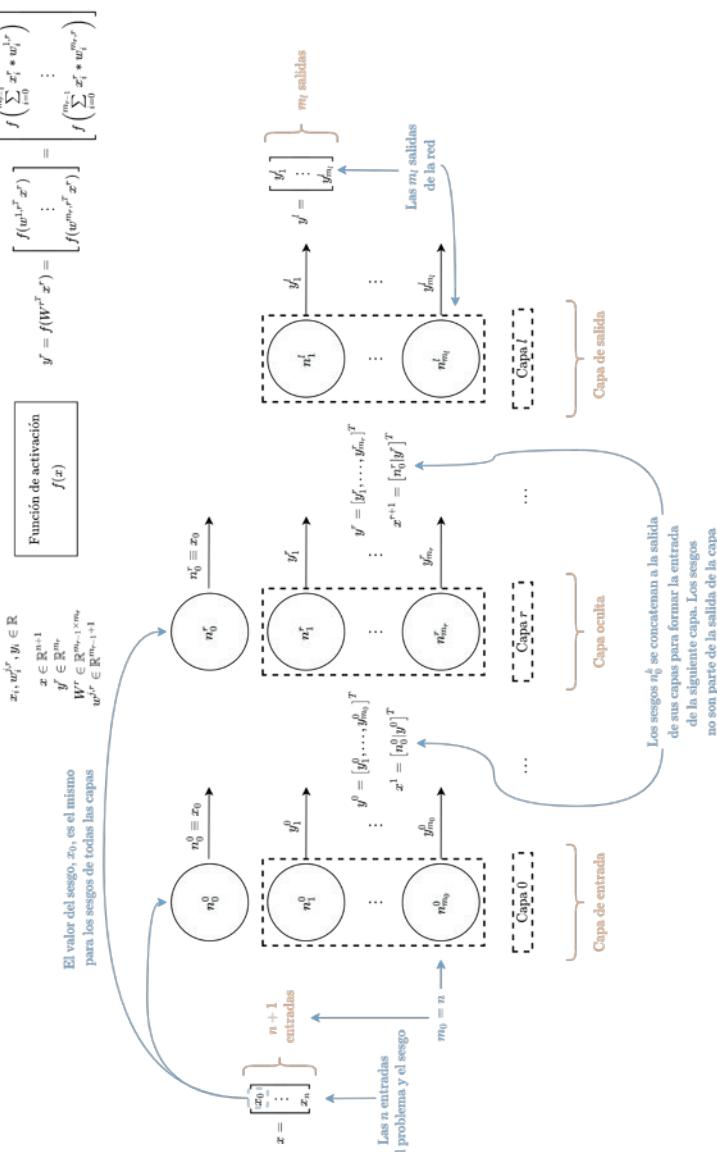
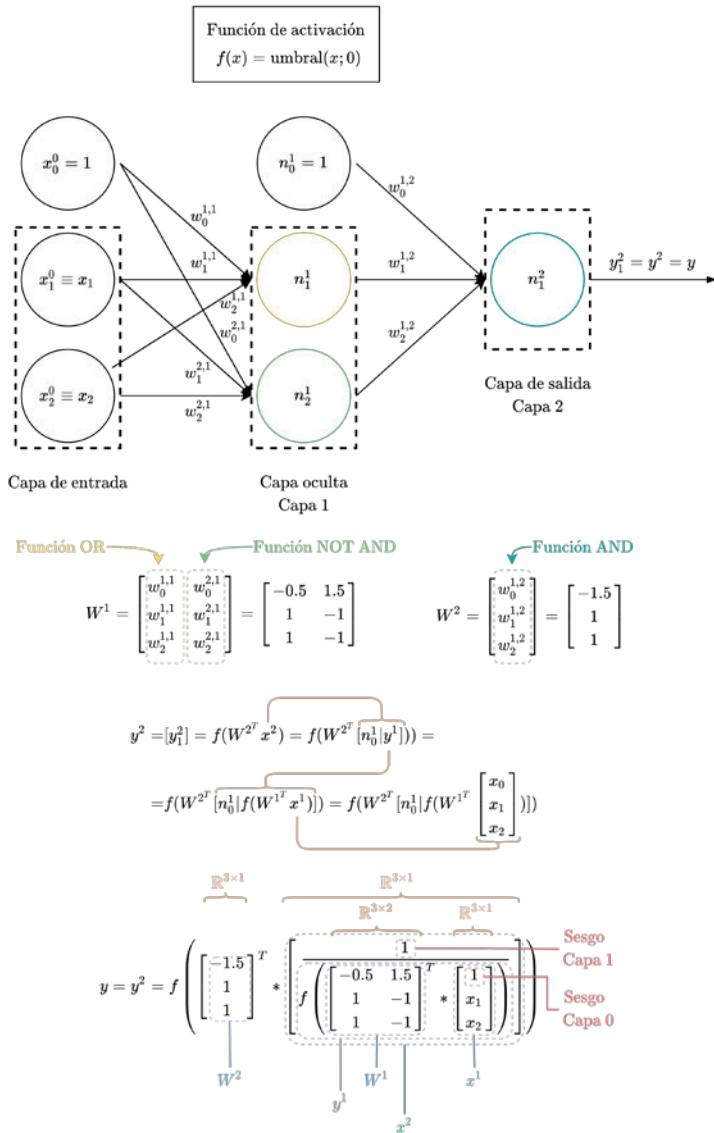


Figura 1.31: Elementos de una red neuronal



**Figura 1.32:** Red neuronal para aproximar la función XOR

### 1.3.2. Aplicaciones

Las redes neuronales cuentan con multitud de aplicaciones, incluidas las propias del perceptrón, puesto que, con una única neurona en la capa de salida, la red neuronal se comporta como un único perceptrón.

Una de las ventajas que ofrecen las redes neuronales es el uso de más de una neurona de salida. Esto permite tratar problemas que un único perceptrón es incapaz de abordar, como la clasificación multiclas. Por ejemplo, para un problema que consista en reconocer un dígito en una imagen se podrían utilizar diez neuronas en la capa de salida. Cada neurona es encargada de detectar si el dígito que tiene asociado es el de la imagen de entrada. Sin embargo, nada garantiza que dos neuronas no se vayan a activar a la vez o que no se active ninguna. Para este tipo de problemas se utiliza la función de activación identidad en la capa de salida y se aplica la función *softmax* al vector de salida. Esta función consiste en una extensión de la función sigmoide, que convierte el vector de salida en una distribución de probabilidad mediante la aplicación

$$\begin{aligned} S : \quad \mathbb{R}^n &\rightarrow \quad \mathbb{R}^n \\ v &\mapsto \quad S(v) \end{aligned} \tag{1.7}$$

donde  $v = [v_1, \dots, v_n]^T$  y  $S = [S_1, \dots, S_n]^T$  con  $S_i = \frac{e^{v_i}}{\sum_{j=1}^n e^{v_j}}$

Gracias a esta aplicación, la salida de cada neurona se puede interpretar como una probabilidad. No obstante, aún no existe una clasificación de la red para la entrada. Es necesario obtener, de este vector, la clasificación final. Para ello, se transforma el vector de probabilidades en una codificación que indique la clase asignada por la red a la entrada, normalmente la clase asociada con la neurona que tiene la mayor probabilidad. Esto se consigue, generalmente, utilizando la codificación *one-hot*, que consiste en asignar el valor 1 a la componente del vector que tiene la mayor probabilidad (si hay más de una se toma cualquiera de ellas) y al resto de elementos el valor 0. En la Figura 1.33 se muestra un ejemplo de la transformación *softmax* y la codificación *one-hot*.

Se observa que el vector de salida,  $y$ , al aplicarle la transformación *softmax*, repre-

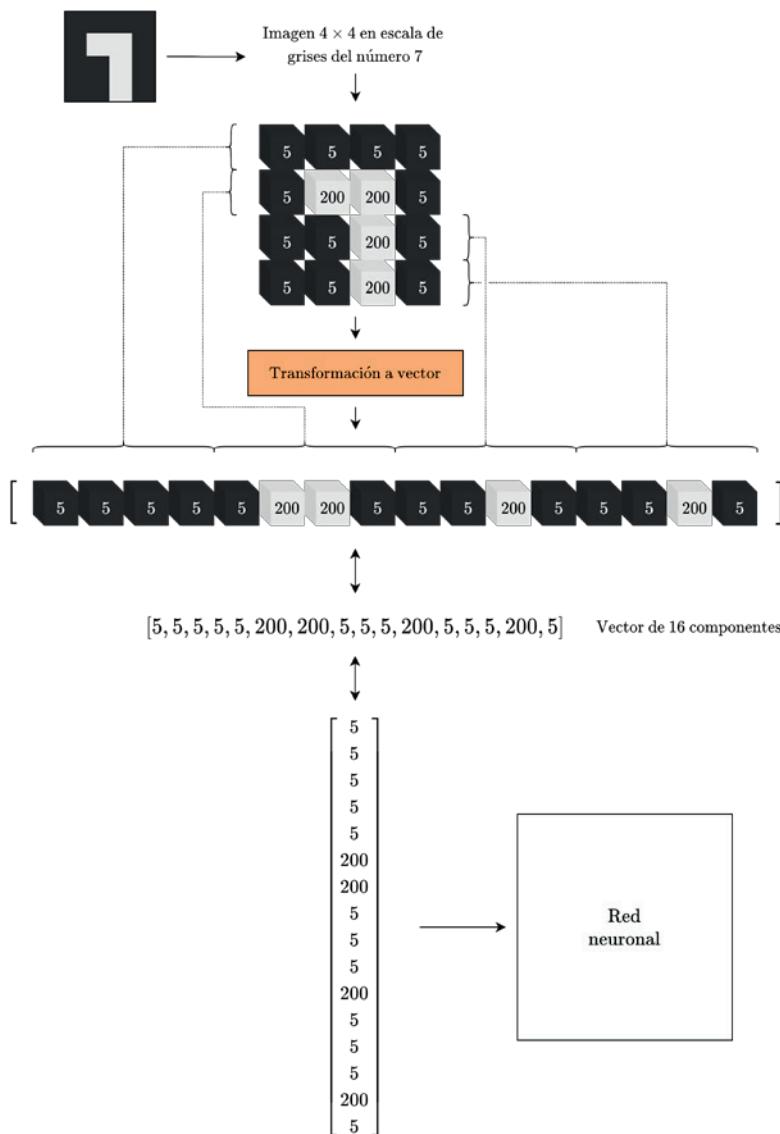
$$y = \begin{bmatrix} 1.2 \\ 0.9 \\ 0.4 \end{bmatrix} \xrightarrow{\text{Softmax}} \begin{bmatrix} 0.46 \\ 0.34 \\ 0.20 \end{bmatrix} \xrightarrow{\text{one-hot}} \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

**Figura 1.33:** Transformación softmax y codificación one-hot

senta una distribución de probabilidad. Si este vector de probabilidades se codifica como *one-hot* se obtiene la clasificación de la entrada, es decir, la clase correspondiente a la posición donde este vector tiene el valor 1. En el caso de la Figura 1.33, la entrada quedaría clasificada como la clase que representa la primera neurona de salida.

Se pueden abordar problemas mucho más complejos que la clasificación multiclase con las redes neuronales. Si se representa una imagen como un vector de píxeles, esta puede ser procesada por una red neuronal, pero se puede ir incluso más allá y generar una imagen como salida. Solo hay que utilizar tantas neuronas en la capa de salida como píxeles se quiera que tenga la imagen. Por ejemplo, si la entrada es una imagen en escala de grises con  $N$  píxeles, se pueden utilizar  $3N$  neuronas en la capa de salida para representar una imagen con tres canales (canal rojo, verde y azul) y con el mismo tamaño que la imagen de entrada. Entonces, se puede utilizar la red para transformar la imagen de entrada en una imagen a color, es decir, para colorear la imagen de entrada.

Como se observa en el ejemplo anterior, una de las limitaciones de las redes neuronales es que tanto su entrada como salida es un vector. Por lo tanto, para poder procesar datos que tengan dimensiones espaciales, como las imágenes, es necesario convertirlos en una representación vectorial. Este proceso se conoce como aplastamiento o *flattening* en inglés (ver Figura 1.34). Esta operación consiste en descomponer las dimensiones espaciales de los datos de entrada para formar un único vector. Por ejemplo, para una imagen en escala de grises con un tamaño de  $28 \times 28$ , 784 píxeles, el vector resultado de la operación *flattening* podría ser un vector de 784 componentes donde los 28 primeros elementos son los 28 píxeles de la primera fila y los últimos 28 son los píxeles de la última fila.



**Figura 1.34:** Ejemplo de la operación *flattening*

### 1.3.3. Limitaciones

Como es sabido, una red puede tener una capa o multitud de ellas. Sin embargo, no por ello hay garantías de que una red con más capas consiga mejores resultados que una red con menos. La **expresividad** de una red es el término con el que se suele hacer referencia a la capacidad que tiene una red de aproximar funciones. Cuanto más complejas sean las funciones que puede aproximar la red mayor será su expresividad.

Las funciones más elementales son las funciones lineales. Estas funciones pueden ser aproximadas con un único perceptrón (una red neuronal con una capa y una neurona en esta capa). Funciones más complejas, como la función XOR, requieren del uso de varias capas para poder aproximarlas, puesto que no son separables linealmente. No obstante, no es el hecho de añadir más capas a una red lo que permite aproximar correctamente estas funciones, sino el uso de funciones de activación que no son lineales. Esto es, su salida no es una combinación lineal de las entradas.

El uso de una función de activación lineal implica que la transformación de los datos que hace la red es equivalente a la que podría hacer una red que cuente con una única capa con tantas neuronas como la capa de salida de la red original. Para ver por qué una función de activación lineal no puede aproximar funciones no lineales hay que tener presente que cualquier función de activación lineal de la forma  $f(x) = mx + b$  es equivalente a introducir esta transformación en la matriz de pesos de la capa y utilizar como función de activación la función identidad,  $f(x) = x$ . La salida de cada capa,  $y^r$ , no es más que la combinación lineal de las entradas y los pesos,  $W^{r^T} * x^r$ . Debido a la propiedad asociativa de la multiplicación, la salida de una red de  $l$  capas es

$$\begin{aligned}
 f(x) &= x \implies y^l = f(W^{l^T} * x^l) = f\left(W^{l^T} * f\left(W^{l-1^T} * x^{l-1}\right)\right) = \\
 &= \dots = \\
 &= f\left(W^{l^T} * f\left(\dots f\left(W^{1^T} * x^1\right)\dots\right)\right) = \\
 &= W^{l^T} * \dots * W^{1^T} * x^1 = \\
 &= (W^{l^T} * \dots * W^{1^T}) * x^1 = W' * x^1
 \end{aligned}$$

Toda la red es equivalente a una matriz de pesos,  $W'$ , con las mismas dimensiones que la matriz de pesos de la capa de salida. Si se multiplica esta matriz por la entrada de la primera capa,  $x^1$ , se obtiene la misma salida que si se aplican todas las operaciones de la red.

Con el uso de funciones de activación no lineales se consigue una mayor expresividad. Gracias al uso de la función de activación umbral, que no es lineal, en el Ejemplo 1.3, se pudo aproximar la función XOR.

La capacidad de las redes neuronales para aproximar funciones queda demostrada con el teorema de aproximación universal [5]. La demostración formal de este teorema está fuera del alcance de este libro. Este teorema afirma que, dada una función continua, siempre se puede conseguir aproximar esta función, con la precisión deseada, utilizando una red neuronal, sin importar lo complicada que sea la función que se desea aproximar, siempre y cuando se utilicen funciones de activación continuas, acotadas y no constantes. Según este teorema, se podría aproximar cualquier función utilizando únicamente una red con dos capas.

Una forma de aproximar una función mediante una red con una única capa oculta es dividiéndola en intervalos. Para el caso de una neurona con función de activación sigmoide, el efecto de la modificación del peso y del *bias* para una única entrada se muestra en la Figura 1.35. Como se observa, a medida que aumenta el peso, la pendiente es más pronunciada y se parece cada vez más a una función umbral; el *bias* provoca que la pendiente se mueva a la izquierda o a la derecha según sea positivo o negativo respectivamente.

Si ahora se utilizan dos de estas neuronas para la capa oculta y se conectan con una tercera neurona, que lo único que hace es cambiar el signo a una de ellas y sumar sus valores, se construye una especie de «pulso» (ver Figura 1.36). Si se desea ampliarlo, solo hay que aumentar el valor de los pesos de la última neurona (ver Figura 1.37).

Con esta forma de trabajar, solo se necesitan dos neuronas para construir el pulso y, dependiendo de la precisión que se desea alcanzar, se pueden añadir estos pares de neuronas para ir逼近ando la función por intervalos cada vez más pequeños, modificando cada pulso para que se aproxime a la forma de la función (ver Figura 1.38). En el caso de aproximar funciones de más de una dimensión, se pueden emplear tantas neuronas de salida como dimensiones haya y utilizar un mayor

número de neuronas en la capa oculta para construir formas más complejas que el pulso de una dimensión, como cilindros en el caso de tres dimensiones.

Una de las conclusiones que se pueden sacar de este teorema es que, dado que se puede interpretar el cerebro humano como una especie de función que recibe unas entradas (sensores) y da unas salidas (sistema motriz), en principio, existe una red neuronal capaz de realizar cualquier tipo de tarea que realizan los humanos, incluso todas y mejor. Por ejemplo, se podría construir una red que, dada una imagen de una radiografía, sea capaz de detectar una enfermedad; una red que, al hablar con humanos, sea capaz de responder, en el mismo idioma, a lo que se le dice. Todas estas tareas son funciones que reciben unas entradas y dan una salida. Y esta ha sido una de las motivaciones que ha empujado a los investigadores a seguir trabajando en este campo: la posibilidad de construir un «cerebro artificial». Sin embargo, es evidente que existen diferencias entre lo que pueden hacer los humanos y lo que una red neuronal es capaz de realizar.

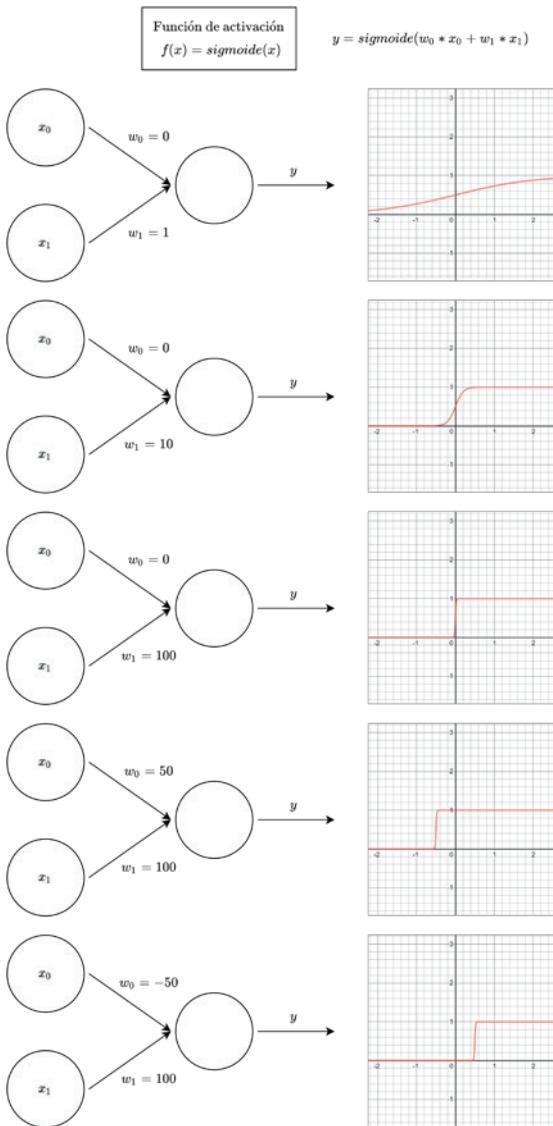
En primer lugar, el teorema no habla de la arquitectura o los pesos que debe tener la red, es un teorema de existencia: garantiza que existe, pero no se sabe cómo dar con ella. Además, esta red puede que necesite un número de neuronas casi infinito o imposible para la tecnología actual. Tampoco hay garantía de que se vaya a encontrar una función exactamente igual a la función objetivo, el teorema garantiza cierta precisión en su aproximación, sin asegurar que ambas vayan a ser idénticas.

Por otro lado, las redes neuronales son tratadas como *cajas negras*, no son modelos **interpretables**, es decir, no se puede comprender por qué, ante una entrada, la red genera cierta salida. Su implantación en entornos críticos de toma de decisiones, donde la vida de una persona esté en juego, como la conducción autónoma, implica siempre ciertos riesgos.

En la Sección 2.6.4 se verán otros problemas a los que se enfrentan estos modelos por la forma en la que se definen los pesos de la red, entre los que destacan su incapacidad para retener las tareas aprendidas cuando es utilizada para una nueva tarea o la poca robustez de estos modelos. Esto último es debido a que, al aproximar la función objetivo, en algunas regiones sí puede tener la forma de la función pero en otras no (ver Figura 1.39). Cuando entran datos que pertenecen a la zona donde no hay una buena aproximación, la salida del modelo no coincide con la esperada. Esto se puede ver fácilmente en el caso de la conducción autónoma: si la red es

entrenada con imágenes de países del norte, al utilizar la red en países más tropicales con una geografía y una delimitación de las carreteras y señales diferente, la red podría tener un comportamiento inesperado. Incluso con una leve modificación del valor de un solo dato de entrada se puede provocar una salida errónea de una red neuronal [6], lo que da una idea de las precauciones que hay que tener a la hora de utilizarlas.

Por último, las neuronas artificiales no imitan la forma de trabajar de las neuronas biológicas. En el cerebro humano existen en torno a  $10^{11}$  neuronas con  $10^4$  conexiones por cada una (números bastante alejados de la capacidad actual de los ordenadores comerciales), con diferentes tipos de neuronas y existiendo una componente temporal, no presente en las redes neuronales artificiales, que rige su funcionamiento. Además, utilizan un mecanismo de aprendizaje, aún por descifrar, mucho más rápido y eficiente que el realizado por las redes neuronales artificiales, cuyo mecanismo de aprendizaje no parece asemejarse al de las neuronas biológicas. A esto hay que añadirle que, a pesar de que las neuronas biológicas procesan la información a una velocidad inferior a la de un ordenador ( $10^{-3}$  segundos entre activación y desactivación) son capaces de resolver tareas extremadamente complejas, imposibles para sus equivalentes artificiales, gracias en parte a su elevado nivel de paralelismo. Valga la analogía de Jeff Hawkins en su libro *On Intelligence*: «Un humano puede realizar tareas considerables en mucho menos tiempo que un segundo. Por ejemplo, podría mostrarles una fotografía y pedirles que indicaran si hay un gato en la imagen. Su labor sería pulsar un botón si hay un gato, pero no hacerlo si ven un oso, un jabalí o un rábano. Esta tarea es difícil o imposible de realizar para un ordenador actual, pero un humano puede hacerlo de forma fiable en medio segundo o menos. Pero las neuronas [biológicas] son lentas, así que en ese medio segundo la información que entra en nuestro cerebro solo es capaz de atravesar una cadena de cien neuronas. Es decir, el cerebro *computa* soluciones a problemas en cien pasos o menos, prescindiendo de cuántas neuronas puedan participar en total. Desde el instante en que la luz entra en nuestro ojo hasta el momento en que pulsamos el botón, podría participar una cadena no más larga de cien neuronas. Cien instrucciones en un ordenador apenas bastan para mover un solo carácter en la pantalla, y ya no digamos para hacer algo interesante».

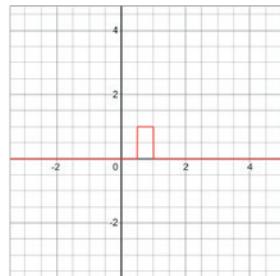
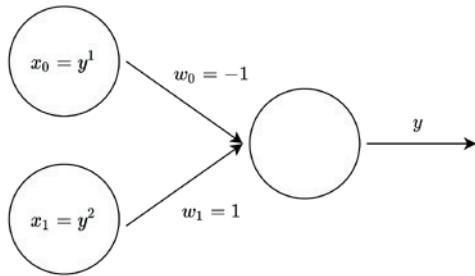
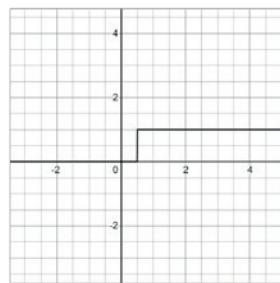
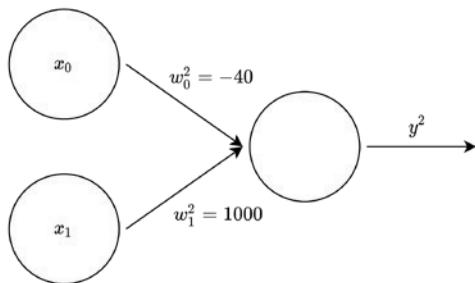
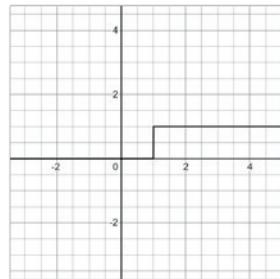
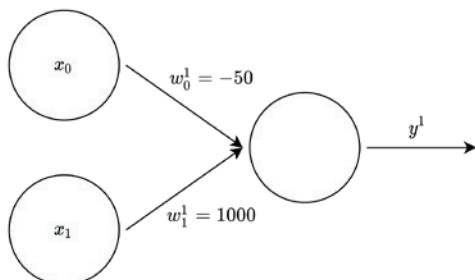


**Figura 1.35:** Efecto del cambio del valor del peso y del *bias* en una neurona con función de activación sigmoide

Función de activación $f(x) = \text{sigmoide}(x)$
--

$$y^i = \text{sigmoide}(w_0^i * x_0 + w_1^i * x_1)$$

$$y = w_0 * x_0 + w_1 * x_1$$

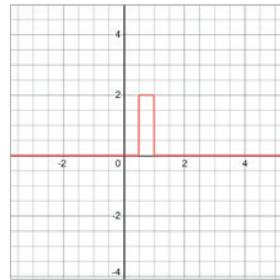
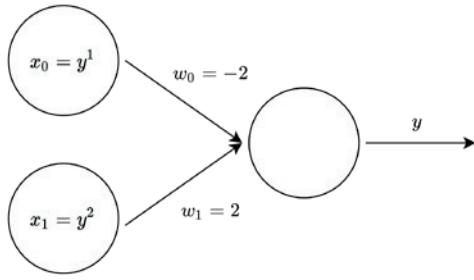
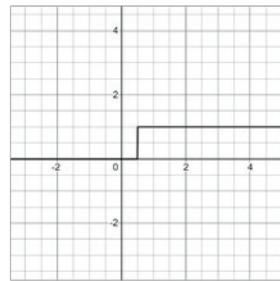
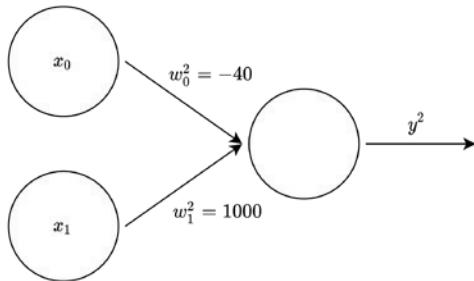
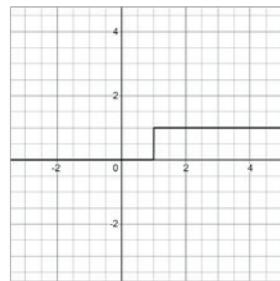
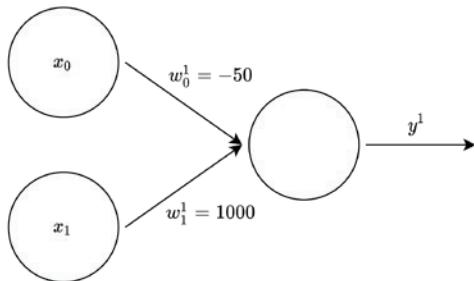


**Figura 1.36:** Construcción de una función «pulso» con una red neuronal

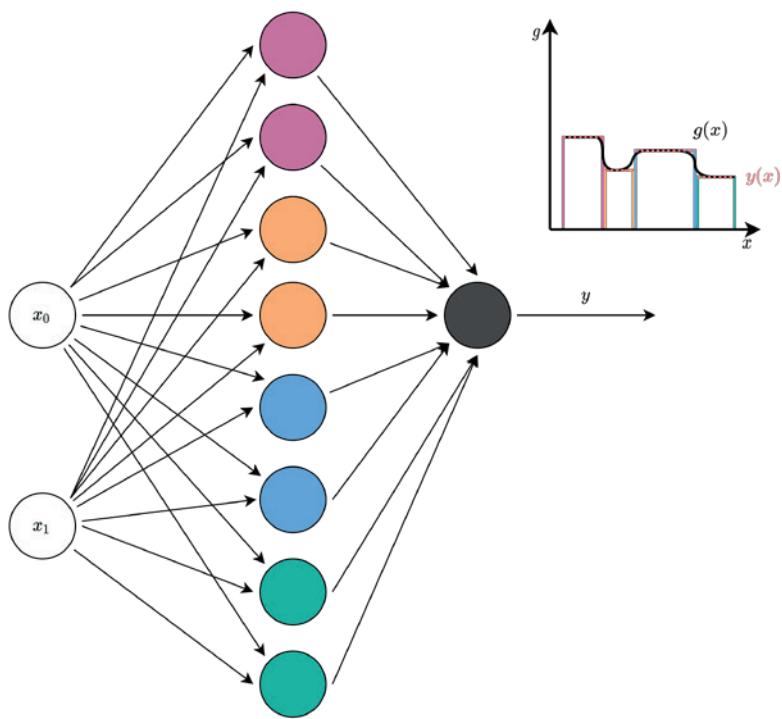
Función de activación $f(x) = \text{sigmoide}(x)$
--

$$y^j = \text{sigmoide}(w_0^j * x_0 + w_1^j * x_1)$$

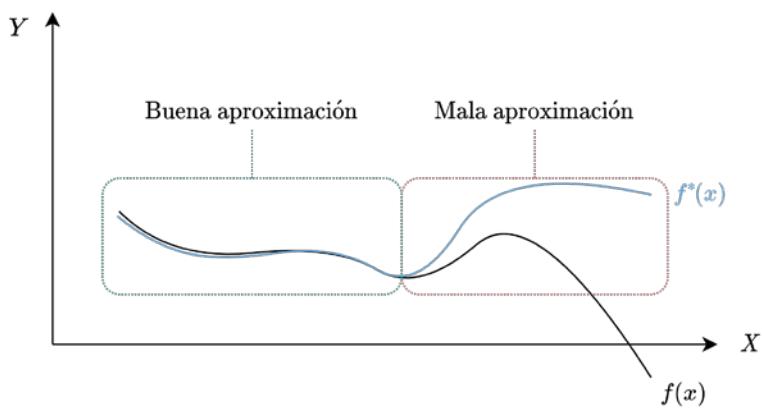
$$y = w_0 * x_0 + w_1 * x_1$$



**Figura 1.37:** Construcción de una función «pulso» ampliado con una red neuronal



**Figura 1.38:** Aproximación de una función con una red neuronal



**Figura 1.39:** Ejemplo de función aproximada correctamente solo en algunas zonas



## Capítulo 2

# Desarrollo de redes neuronales

Una vez se ha estudiado los componentes, aplicaciones y limitaciones de las redes neuronales, toca centrarse en su desarrollo.

Para crear una red neuronal es necesario definir primero su arquitectura. Esta etapa de diseño se considera más un arte que una ciencia dado que, dependiendo del problema que se esté tratando, habrá arquitecturas que funcionen mejor que otras; suele ser un proceso de prueba y error. La estrategia más habitual consiste en utilizar alguna de las arquitecturas populares por sus resultados en problemas genéricos. Se presupone que las arquitecturas que funcionan en estos problemas lo harán también en el problema que se esté abordando, a priori más sencillo que el original para el que se diseñó la arquitectura.

Otro elemento clave en la creación y uso de las redes neuronales son los datos. Todos los problemas que son abordados con redes neuronales se reducen a un problema de aproximación de funciones. La función que se desea aproximar se describe en base a una serie de puntos de ejemplo, como en el problema de la función XOR, donde se dispone de todas las entradas y salidas que se esperan obtener para cada entrada. Sin embargo, la mayoría de las veces, no se conoce ni la función que se quiere aproximar ni el valor de la función sobre todos los puntos de su dominio sino, como mucho, el valor de la función sobre algunos ejemplos. Piense el lector en un sistema de detección de personas en imágenes. Es imposible recopilar todas las entradas y salidas de este sistema. Normalmente se trabaja con una muestra de estos datos.

Una arquitectura junto con un conjunto de datos forman el punto de partida des-

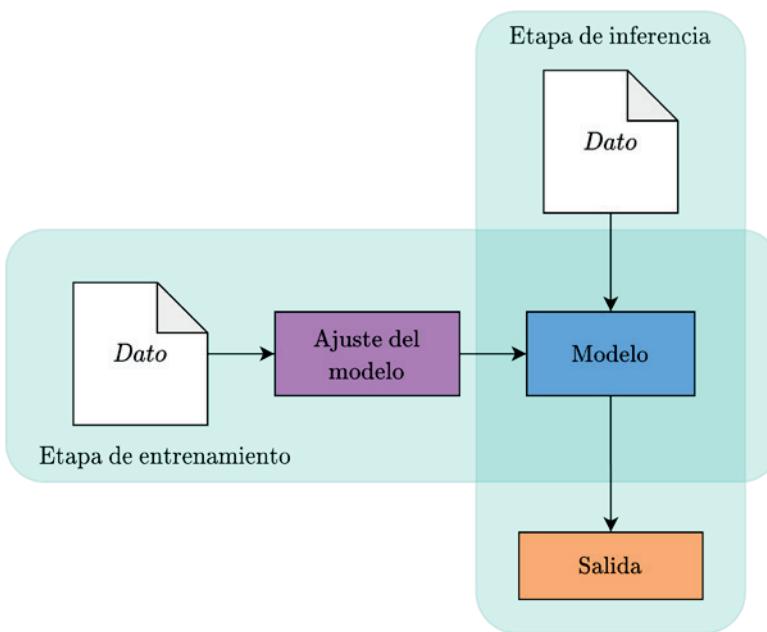
de el que desarrollar la red neuronal que resuelva el problema. En esta sección se estudiará en detalle todo lo relacionado con la creación y uso de las redes neuronales.

## 2.1. Etapa de entrenamiento vs. inferencia

Cuando se trabaja con redes neuronales, o cualquier modelo de *machine learning*, se suele hacer una distinción entre la obtención del modelo y el uso del modelo. Estas dos etapas son conocidas como etapa de entrenamiento y etapa de inferencia (ver Figura 2.1):

- **Etapa de entrenamiento.** En esta etapa la red aún no tiene los pesos definidos. Se utilizan los datos disponibles de forma que la red ajuste lo mejor posible la función que se desea aproximar. El ajuste de los pesos puede ser de forma manual, aleatoria o automática mediante algún **algoritmo de entrenamiento**. En el caso de las redes neuronales, se suele denominar a esta etapa como **etapa de aprendizaje**. El resultado de esta etapa es una red con unos pesos definidos.
- **Etapa de inferencia.** Cuando no se actualizan los pesos de la red se dice que la red se encuentra en la etapa de inferencia. Esta etapa es útil para validar el entrenamiento de la red y decidir si seguir o no en base a los resultados que obtiene. Una de las ventajas de las redes neuronales, como es sabido, es que permiten obtener la salida para cualquier valor de entrada; no se limitan a los datos que ha utilizado durante la etapa de entrenamiento. Por lo tanto, es posible darle a la red datos que nunca ha visto para hacer predicciones sobre estos. El resultado de esta etapa es, únicamente, la salida de la red para los datos de entrada que se le suministren.

La primera de las etapas siempre es la de entrenamiento. Sin unos pesos definidos no se puede utilizar la red. El modelo que se haya escogido para el problema en cuestión será entrenado con los datos disponibles y, una vez se considere finalizado el entrenamiento, se pasará a la etapa de inferencia, donde el modelo no es modifi-



**Figura 2.1:** Etapa de entrenamiento vs. inferencia

cado y, simplemente, obtiene la salida para los datos, ya sean del conjunto de datos que se dispone o nuevos.

Siempre es posible volver a la etapa de entrenamiento para mejorar el modelo y realizar un proceso cíclico de mejora conforme se obtienen más datos para su entrenamiento.

## 2.2. Paradigmas de aprendizaje

Durante la etapa de entrenamiento se pueden ajustar los pesos de la red de muchas formas diferentes. En *machine learning*, se suele utilizar algún algoritmo de

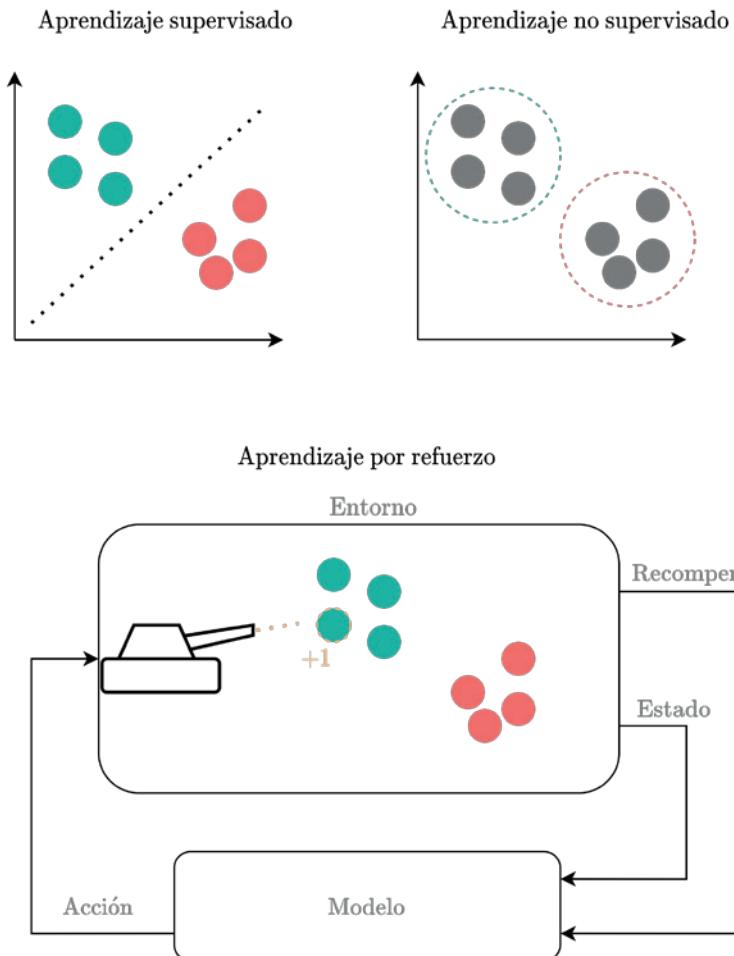
aprendizaje que se encarga de ajustar los pesos del modelo utilizando un conjunto de datos. En base al tipo de dato que se utilice para ajustar el modelo se distinguen tres paradigmas de aprendizaje, el aprendizaje supervisado, no supervisado y por refuerzo (ver Figura 2.2):

- **Aprendizaje supervisado.** Cuando los datos tienen asociados una salida esperada, como en el ejemplo de la función XOR, se está ante un problema de aprendizaje supervisado. Se dispone de información que permite *supervisar* el entrenamiento del modelo. Según la diferencia entre la salida del modelo y el valor esperado se puede ajustar los pesos de la red. Dentro de este tipo de problemas se incluye a la clasificación y a la regresión que se estudió en la Sección 1.2.2. El principal problema del aprendizaje supervisado es la falta de datos para entrenar correctamente al modelo. Esto es debido a que, para la mayoría de los problemas de aprendizaje supervisado, la salida esperada para cada dato de entrada debe ser definida de forma manual, lo que ralentiza y encarece la elaboración de un buen conjunto de datos.
- **Aprendizaje no supervisado.** Si los datos no tienen una salida asociada, se trata de un problema de aprendizaje no supervisado. No existe información de cómo de bien lo hace la red. En estos casos se suele utilizar la red neuronal para analizar los datos y buscar patrones que permitan agruparlos, es lo que se conoce como *clustering*. Al no requerir que los datos tengan una salida asociada, los problemas de aprendizaje no supervisado suelen tener muchos más datos disponibles, aunque las aplicaciones no son tan amplias como las del aprendizaje supervisado.
- **Aprendizaje por refuerzo.** En este paradigma de aprendizaje, la red es tratada como un agente que interactúa con un entorno. En lugar de disponer de un conjunto de datos como en los casos anteriores, el agente envía acciones al entorno y de este recibe el estado en el que se encuentra el agente después de ejecutar la acción en el entorno y una *recompensa* que simboliza lo bien o mal que ha sido aplicar la acción enviada por el agente en el estado en el que se encontraba. Como si de un videojuego se tratase, el agente debe ajustar la red en base al *feedback* que recibe del entorno para ir mejorando. Este tipo de aprendizaje es con el que más se identifican a los humanos: los humanos son los agentes y el mundo es el entorno, en el que se ejecutan acciones y del que se recibe una retroalimentación en base a las acciones realizadas. A pesar de que pueda parecer el más idóneo al no tener que etiquetar los datos de forma manual como en el

aprendizaje supervisado, su principal desventaja es la falta de retroalimentación para cualquier estado posible en el que se encuentre el agente. No siempre es evidente el valor de la retroalimentación para cualquier estado. Por ejemplo, si en mitad de un laberinto, la red gira a la izquierda, en ese instante, no se sabe si recompensar o castigar la acción hasta que no se llegue al final del laberinto; puede incluso que no sea ni relevante en el resultado final.

Existe la posibilidad de modelar un problema bajo diferentes paradigmas. Por ejemplo, un problema de clasificación puede abordarse desde los tres paradigmas. El aprendizaje supervisado es el más evidente, pero con aprendizaje por refuerzo se podría interpretar una clasificación correcta de la red como una acción a recomendar y la clasificación incorrecta como una acción a penalizar; el entorno sería los propios datos del problema. También es posible abordar el problema desde la perspectiva del aprendizaje no supervisado. Si la clasificación de los datos está relacionada con la dispersión que presentan, se pueden encontrar estas agrupaciones para utilizarlas como la clasificación en las diferentes clases del problema.

Estos tres paradigmas son los más genéricos que existen, aunque no son los únicos. Sin embargo, queda fuera del alcance de este libro el estudio de otros paradigmas.



**Figura 2.2:** Aprendizaje supervisado, no supervisado y por refuerzo

## 2.3. Obtención y preparación de los datos

Como se ha visto en secciones anteriores, cuando se utiliza una red neuronal, siempre se manejan unos datos de entrada, incluso se consideran una capa de la red. Ya sea para verificar que el modelo funciona correctamente o para ajustar sus pesos, se va a necesitar un conjunto de datos para trabajar con las redes.

Los datos que se vayan a recopilar deben ser representativos del problema y recoger correctamente su casuística. Es conocida la polémica reciente con Google cuando su aplicación, Google Fotos, clasificaba imágenes de personas negras como gorilas. Esto se debe a que su sistema fue diseñado con imágenes de personas blancas mayoritariamente y, por lo tanto, este sesgo fue adquirido por la red neuronal, que fallaba al clasificar las personas de otro color de piel. Como curiosidad, la decisión que adoptaron fue la de eliminar la clase gorila de la aplicación. Este es solo un ejemplo de los desafíos a los que hay que enfrentarse al trabajar con las redes neuronales, en parte por su poca interpretabilidad, que obliga a prestar mucha atención a la hora de construir el conjunto de datos para no introducir sesgos que provoquen comportamientos poco éticos, discriminatorios o incluso peligrosos. Por ejemplo, si un coche autónomo es entrenado con un conjunto de datos construido con imágenes de carreteras de países cálidos, cuando se utilice en países del norte, no se sabrá el comportamiento que tendrá el sistema al conducir por paisajes nevados.

La fase de obtención de los datos no solo es clave para eliminar estos sesgos, sino esencial para conseguir buenos resultados. Como se verá en el Capítulo 4, la llegada de la denominada era del *big data* ha propiciado, junto a otros factores, el éxito de las redes neuronales. La obtención de un conjunto de datos suele ser una tarea laboriosa donde se clasifica, de forma manual, la mayoría de los datos para, después, entrenar la red. Hoy día existen empresas que se dedican en exclusiva a clasificar/etiquetar los datos que otras empresas quieren utilizar para entrenar una red. Otras compañías inventan ideas más creativas. Tesla, por ejemplo, cuenta con conexión a Internet en todos sus vehículos, donde su sistema de conducción autónoma está siempre procesando los datos de entrada, aunque el conductor lo desactive. Cuando la decisión del sistema no coincide con la que toma el conductor, se envían los

datos de entrada del sistema a los servidores de Tesla para ser etiquetados y entrenar con ellos a la red neuronal. Es importante notar que estos datos son los más escasos, puesto que, en caso contrario, la red hubiera actuado bien al haber sido una entrada similar a las utilizadas durante el entrenamiento. Muchas veces, la diferencia entre un sistema u otro radica en el conjunto de datos que utilizan: contra más situaciones anómalas recojan, más probabilidades de que el sistema actúe correctamente cuando se den situaciones atípicas.

En el caso de que no se puedan obtener más datos, bien por ser de difícil acceso o carecer de recursos, se pueden utilizar alguna técnica de aumento de datos para crear nuevos datos *sintéticos* con los que entrenar la red. Por ejemplo, en el caso de imágenes, se les puede aplicar multitud de alteraciones como cambios de brillo o contraste, creando una nueva imagen que mantiene la misma salida esperada que la imagen original. Otra opción consiste en acudir a conjuntos de datos públicos que estén disponibles en Internet y utilizarlos para aumentar el conjunto de datos original. También es posible descargar datos que estén accesibles en cualquier sitio web, siempre y cuando se esté seguro de que se está respetando la normativa en relación a los derechos de autor o que el dueño de los datos ha autorizado su uso.

Una vez tenemos un conjunto de datos se puede entrenar una red neuronal para resolver el problema. Normalmente, el conjunto de datos original o *data set* es dividido en tres conjuntos, entrenamiento (*train* en inglés), validación (*validation* en inglés) y prueba (*test* en inglés):

- **Conjunto de entrenamiento.** Los datos de este conjunto serán utilizados por la red para ajustar los pesos de sus neuronas durante la etapa de entrenamiento.
- **Conjunto de validación.** Para saber si la red tiene un buen comportamiento con datos que nunca ha visto se utiliza este conjunto. No se entrena la red con estos datos, sino que se usan para evaluar la red durante el entrenamiento y tomar decisiones sobre este en base a los resultados.
- **Conjunto de prueba.** Este conjunto de datos tiene el mismo objetivo que el conjunto de validación: evaluar el rendimiento del modelo. A diferencia del conjunto de validación, el conjunto de pruebas no se utiliza durante el entrenamiento de la red, sino al finalizar este para estimar el comportamiento que tendrá la red durante la etapa de inferencia.

Dado que los conjuntos de datos para el entrenamiento de redes neuronales deben

ser especialmente grandes, suele ser habitual dejar en torno al 5 % de los datos para validar y otro 5 % para pruebas, dejando el resto de los datos para entrenar la red. No obstante, hay que tener en cuenta que el tamaño de los conjuntos de validación y prueba indica la granularidad con la que se van a detectar mejoras de la red. Por ejemplo, si se dispone de 100 datos en estos conjuntos, solo se va a detectar mejoras por encima del 1 % siendo imposible saber si dos redes neuronales son diferentes por menos de ese porcentaje.

Al contrario que en otros modelos de *machine learning*, no se suele utilizar técnicas de división del conjunto de datos como *cross-validation* o similares. Los conjuntos de datos suelen ser lo bastante grandes y el entrenamiento de las redes con estas técnicas lo suficientemente costosos como para que suponga algún tipo de beneficio.

Por el hecho de manipular conjunto de datos enormes se puede perder el control de estos y tener problemas con datos repetidos o erróneos. Se debe comprobar siempre la integridad de los datos con los que se trabaja, así como de que no aparezcan datos repetidos. Si la mitad del conjunto de validación es idéntico, con que el modelo lo haga bien con un dato, dará la sensación de que funciona correctamente la mitad del tiempo. Para evitar este tipo de problemas es importante tener un control de los datos e incluso revisarlos o extraer ciertas estadísticas de estos, o de los conjuntos globalmente, para estar seguros de que no presentan problemas. Una forma de no correr riesgos consiste en hacer la división de los datos sin manipular los datos, sino las rutas de estos en el sistema de ficheros. Con ello se garantiza que no se dupliquen, pierdan o se corrompan y se ahorra el movimiento de grandes cantidades de datos.

El almacenamiento de los datos debe ofrecer ciertas garantías de seguridad ante posibles daños físicos en los sistemas de almacenamiento. Es común contar con copias en la nube para prevenir posibles pérdidas. Dado que los conjuntos de datos suelen ser bastantes grandes y se pretende que duren en el tiempo, es conveniente almacenarlos con algún formato conocido y, si es posible, que ofrezcan algún tipo de compresión para ahorrar espacio. El formato CSV suele ser el más utilizado para datos tabulados y JPG para las imágenes, ya que cuenta con una técnica de compresión.

No hay que olvidar que la red trabaja con valores reales como entrada. No obstante, los datos no tienen por qué ser valores reales. Sin embargo, siempre hay que

transformar los datos a una representación en la que se usen valores reales. Algunos datos como las imágenes no necesitan mucha manipulación (los valores de los píxeles se pueden representar como un valor real), pero otros sí requieren de técnicas más elaboradas para su representación. Por ejemplo, en el caso de contar con un sistema que sea capaz de distinguir perros y gatos en imágenes, se puede guardar la clase de cada imagen utilizando una codificación *one-hot* con dos componentes en el vector (uno para la clase perro y otro para la clase gato) y, según la clase a la que pertenezca la clase de entrada, se le asignará el valor 1 en la posición correspondiente. Un problema de la codificación *one-hot* es que al utilizar multitud de clases puede que se necesite un vector demasiado grande. Por ejemplo, si se quiere detectar si un *tweet* habla bien o mal de un determinado producto, no se puede codificar el *tweet* con un vector *one-hot* para todos los caracteres posibles que puede tener. Además, cada *tweet* puede tener un número de caracteres variables, por lo que es impracticable utilizar la codificación *one-hot* en estos casos. Por suerte, se han elaborado representaciones más complejas, y a la vez mejores, para representar textos. Es lo que se conoce como *word embedding*. Son modelos que buscan representar palabras como vectores de un espacio multidimensional donde palabras similares ocupan posiciones cercanas de forma que esta nueva representación facilite la resolución del problema a la red. Para el caso de datos de audio se suele trabajar con una imagen del espectrograma y, por lo tanto, se convierte el problema de procesamiento de audio en un problema de procesamiento de imágenes.

## 2.4. Métricas de evaluación

No se puede estar seguro de que la red funcione correctamente sin alguna forma de evaluar su rendimiento. Este es el objetivo de las métricas de evaluación. Además de ayudar a monitorizar el entrenamiento como se estudiará en la Sección 2.8, las métricas de evaluación proporcionan una medida objetiva que permite comparar redes con pesos o arquitecturas diferentes.

Las métricas de evaluación dan una estimación, en base a diferentes criterios cada una, de cómo de bien funciona una red. Evidentemente, para poder hacer esta estimación, es necesario conocer la clase del dato de entrada, se necesita la **salida esperada** para la entrada que se le ha dado a la red y así poderla comparar con la

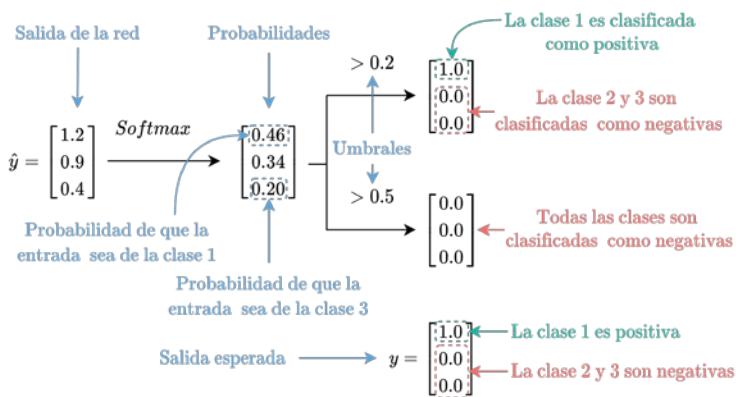
salida obtenida. Por lo tanto, no se pueden utilizar métricas de evaluación durante la fase de inferencia, ya que no se dispone de la salida esperada para los datos de entrada. Por el contrario, en los conjuntos de datos de entrenamiento, validación y prueba sí se dispone de estas salidas esperadas.

En principio, nada impide medir el rendimiento sobre los conjuntos de entrenamiento, validación y prueba. No obstante, lo normal es evaluar la red sobre los conjuntos de prueba y validación. En este último se evalúa varias veces, según se estipule, durante el entrenamiento, para ir «midiéndole el pulso» a la red. Si se detecta que las métricas no mejoran o empiezan a empeorar se puede parar el entrenamiento e intentar una estrategia diferente o buscar si hay algún error en el código. La evaluación sobre el conjunto de entrenamiento se suele utilizar para compararla con la de validación, ya que se espera que ambas sean similares. Si ocurre lo contrario, suele ser síntoma de algún problema. Los problemas derivados del entrenamiento se estudian en detalle en la Sección 2.6.4. En caso de que el entrenamiento necesite de un elevado tiempo de procesamiento se puede obviar la evaluación sobre el conjunto de entrenamiento y guiarse solo por las métricas del conjunto de validación.

No siempre se utilizan las mismas métricas de evaluación. Estas varían en función del tipo de problema en el que se esté trabajando. Las métricas más populares son las relacionadas con los problemas de clasificación. Como es sabido, un problema de clasificación consiste en asignar, a una entrada, una de las clases del problema. En el caso de las redes neuronales, esto se consigue aplicando la operación *softmax* seguida de la codificación *one-hot* a la salida de la red. Dependiendo de la posición donde se encuentre el valor 1 en el vector de salida se sabe la clase que le ha asignado la red a la entrada. El problema de este enfoque radica en que se puede acabar clasificando a la entrada como perteneciente a una clase a pesar de que el valor de probabilidad sea demasiado bajo. Existe una alternativa, que es la utilizada a la hora de evaluar las redes, para corregir este problema, que consiste en emplear un valor de **umbral**, 0.5 por ejemplo. Si el valor de mayor probabilidad de la salida supera este umbral, se le asigna la clase asociada. Ahora bien, puede que este valor sea inferior o que, a pesar de ser superior, no se corresponda con la clase esperada de la entrada.

Para poder comparar correctamente la salida de la red y la salida esperada se distingue entre clase positiva y negativa. En el caso de que el problema tenga más de dos

clases, se suele extraer las métricas considerando cada una de ellas como positivas y al resto como negativas, posteriormente se pueden promediar estas métricas para estimar el comportamiento general de la red para todas las clases. En lo relativo a la salida esperada, esta consiste en un vector en codificación *one-hot* en el que el valor 1 indica la clase que se sabe que corresponde a la entrada, es decir, la clase de esa posición es la clase positiva para esta entrada, el resto de las clases son consideradas clases negativas para la entrada actual. Por otro lado, en la salida de la red, si el mayor valor de probabilidad supera el umbral, se considera como clase positiva la clase asociada a la posición de este valor; el resto de las clases son consideradas clases negativas para la entrada. Si, por el contrario, no supera el umbral de probabilidad, todas las clases se consideran negativas para la entrada. En la Figura 2.3 se muestra un ejemplo de la clasificación de la red para dos valores de umbrales diferentes y la salida esperada.



**Figura 2.3:** Efecto en la clasificación de diferentes valores de umbrales

En base a la comparación entre las clases positivas y negativas obtenidas por la red y las esperadas, se obtienen las cuatro métricas básicas de un problema de clasificación:

- Verdaderos positivos, *true positives* en inglés o **TP** por sus siglas. Si la clase asig-

nada por la red como positiva coincide con la clase positiva esperada se cuenta como un verdadero positivo.

- Verdaderos negativos, *true negatives* en inglés o **TN** por sus siglas. Si la/s clase/s asignada/s como negativa/s por la red coinciden con la/s clase/s negativa/s esperada/s se cuenta como un verdadero negativo.
- Falsos positivos, *false positives* en inglés o **FP** por sus siglas. Cuando la clase positiva predicha por la red no coincide con la esperada es un falso positivo.
- Falsos negativos, *false negatives* en inglés o **FN** por sus siglas. Cuando la/s clase/s negativa/s de la red no coinciden con la/s clase/s negativas esperada/s se está ante un falso negativo.

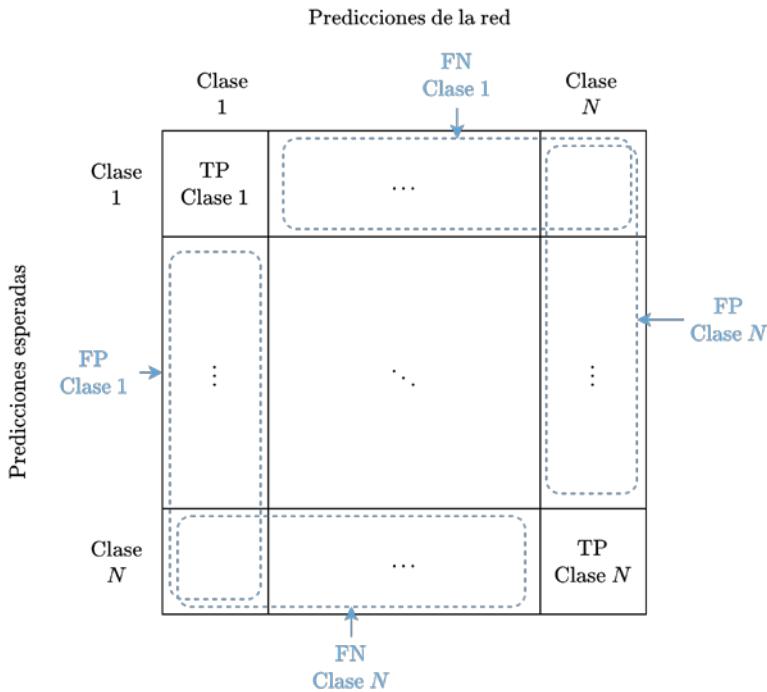
Estas cuatro métricas se suelen agrupar en lo que se conoce como **matriz de confusión**. Se trata de una matriz que se utiliza como un contador, donde cada fila corresponde a las clases esperadas para cada entrada y las columnas se corresponden con las clases predichas por el modelo. Para cada dato de entrada, en base a la clase esperada y la clase predicha por la red, se incrementa la celda correspondiente. La matriz de confusión se puede construir para todas las clases del modelo como se muestra en la Figura 2.4. Esta es la metodología a seguir si se utiliza la codificación *one-hot* a la salida de la red; si, por el contrario, se aplica un umbral a la salida, se usa una matriz de confusión para cada una de las clases, donde el resto de las clases son consideradas negativas. Esta última es la opción más habitual y es la mostrada en la Figura 2.5.

A partir de la matriz de confusión obtenida para cada una de las clases se derivan la mayoría de las métricas más utilizadas:

- Exactitud o **accuracy** en inglés. Esta métrica da el ratio de datos clasificados correctamente (tanto para la clase positiva como negativa):

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- Exhaustividad o **recall** y **sensitivity** en inglés. Con esta métrica se pone el foco en lo bien que lo hace el modelo con los datos de la clase positiva. Da el ratio de datos de la clase positiva que han sido bien clasificados:



**Figura 2.4:** Matriz de confusión para  $N$  clases

$$\text{Recall} = \frac{TP}{TP + FN}$$

- Precisión o ***precision*** en inglés. Esta métrica es utilizada para saber el ratio de datos que son realmente de la clase positiva respecto del total de ejemplos que la red ha clasificado como positivos:

$$\text{Precision} = \frac{TP}{TP + FP}$$

		Predicciones de la red	
		Clase Positiva	Clase Negativa
Predicciones esperadas	Clase Positiva	TP	FN
	Clase Negativa	FP	TN

**Figura 2.5:** Matriz de confusión considerando solo las clases positivas y negativas

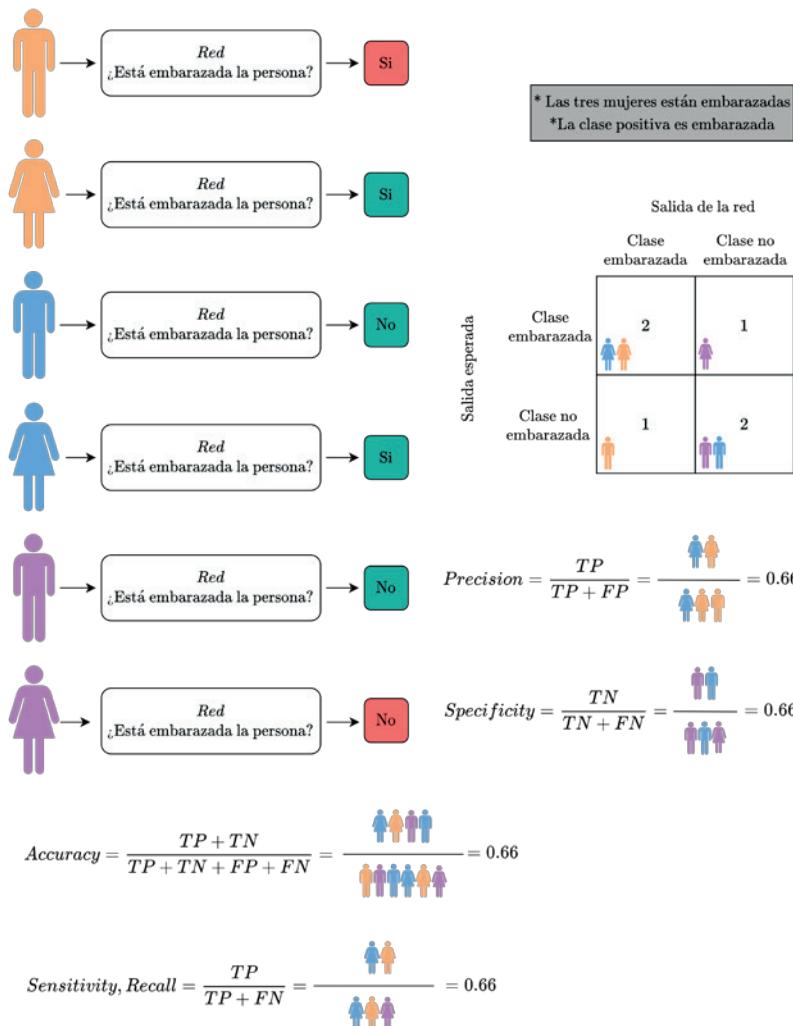
- Especificidad o **specificity** en inglés. Similar a *recall* solo que esta métrica se centra en los datos de la clase negativa:

$$\text{Specificity} = \frac{TN}{TN + FN}$$

En la Figura 2.6 se muestra un problema de ejemplo sobre el que se calculan estas métricas.

El problema principal de estas métricas es que se deben definir un valor de umbral para asignarle a cada dato de entrada la clase positiva o negativa. Dependiendo del valor de umbral, habrá métricas que salgan más beneficiadas que otras. La elección de un umbral no es tarea fácil. Un umbral bajo permite detectar todos los positivos y, por tanto, la red contará con una sensibilidad alta, aunque pueden aparecer falsos positivos. Si, por el contrario, se utiliza un umbral alto, los falsos positivos acaban por desaparecer a costa de que se puedan perder positivos verdaderos. En este caso el sistema tendrá una especificidad alta. Estas dos métricas son incompatibles y el valor del umbral ayuda a elegir el comportamiento del modelo más adecuado para cada problema.

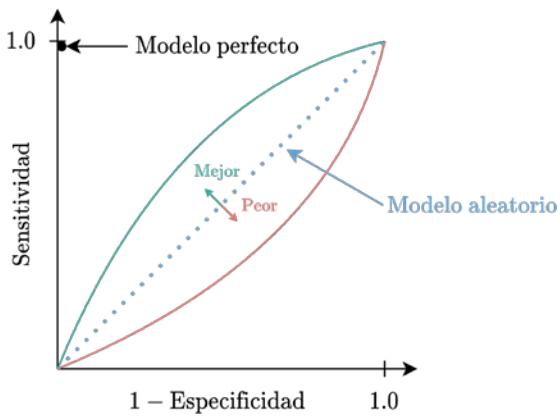
Si se quiere comparar dos modelos diferentes se pueden utilizar métricas que no dependan del valor de umbral elegido, como el **área bajo la curva ROC**. Esta métrica



**Figura 2.6:** Métricas para un problema de clasificación de personas según están embarazadas o no

consiste en enfrentar los valores de sensibilidad y especificidad para, a continuación, medir el área bajo esta curva (AUC por sus siglas en inglés) como métrica

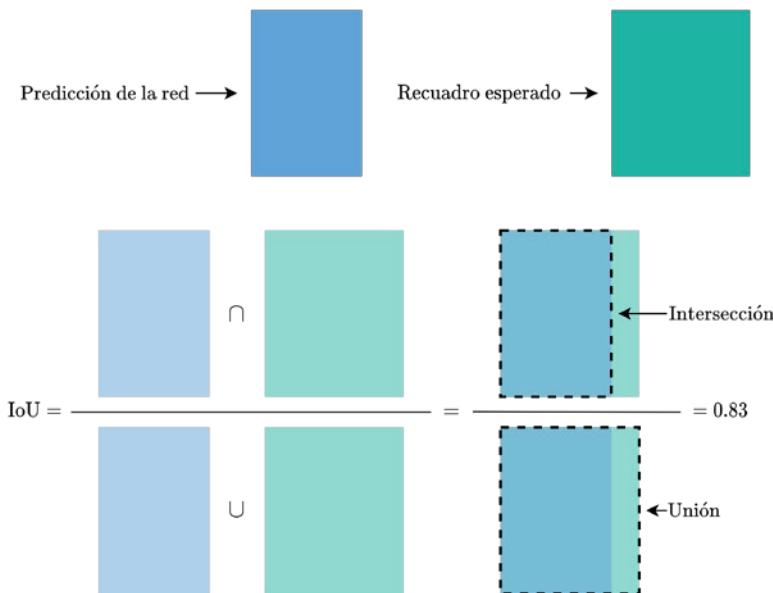
que define el rendimiento general del modelo (ver Figura 2.7). Se considera que un clasificador es perfecto si su área es igual a la unidad mientras que un clasificador cuya curva se sitúa cerca de la diagonal, o por debajo, se considera que no es útil, puesto que se confunde más que acierta. La diagonal se suele interpretar como el rendimiento de un modelo que realiza predicciones aleatorias.



**Figura 2.7:** Curva ROC

Cuando no se está ante un problema de clasificación suelen utilizarse otras métricas, más específicas del problema en cuestión. Por ejemplo, si se está trabajando en un problema de detección de objetos, la salida de la red suele ser un conjunto de recuadros que delimitan los objetos que la red ha detectado en la imagen. En este problema se suele emplear una métrica conocida como ***intersection over union***, o IoU por sus siglas, que consiste en calcular el ratio entre la intersección de un recuadro predicho por la red y el recuadro esperado y la unión de ambos (ver Figura 2.8).

Se pueden encontrar multitud de ejemplos de métricas orientadas a determinados problemas. Dependiendo del problema que se esté abordando será necesario buscar en la literatura las métricas más utilizadas para poder comparar la solución propuesta con otras en igualdad de condiciones.

**Figura 2.8:** Métrica IoU

## 2.5. Estimación del error

El objetivo de la fase de entrenamiento de una red neuronal es establecer los pesos de esta utilizando el conjunto de datos de entrenamiento. Para modificar los pesos de una red se utiliza una medida que indica la desviación del valor dado por la red al valor esperado para los datos del conjunto de entrenamiento. Este valor es conocido como el **error** de la red. Por otro lado, se puede utilizar una estimación de cómo de malo es conseguir determinado valor de error, es lo que se conoce como **pérdida** o *loss* en inglés. La función que, dada la salida de la red y la salida esperada, nos devuelve el valor de error o pérdida se llama **función de error**, **función de pérdida** o **función de coste**.

A la hora de elegir la función de error hay que tener en cuenta el problema que se está abordando, ya que algunas funciones son más indicadas que otras según el problema. Las más utilizadas son las siguientes:

- **Error cuadrático medio** (ECM) o *mean square error* en inglés. Normalmente, se utiliza en problemas de regresión. Consiste en la media de las diferencias al cuadrado entre la salida esperada y la salida obtenida. Para un conjunto de  $n$  datos y de dimensión  $m$ , el ECM se calcula como

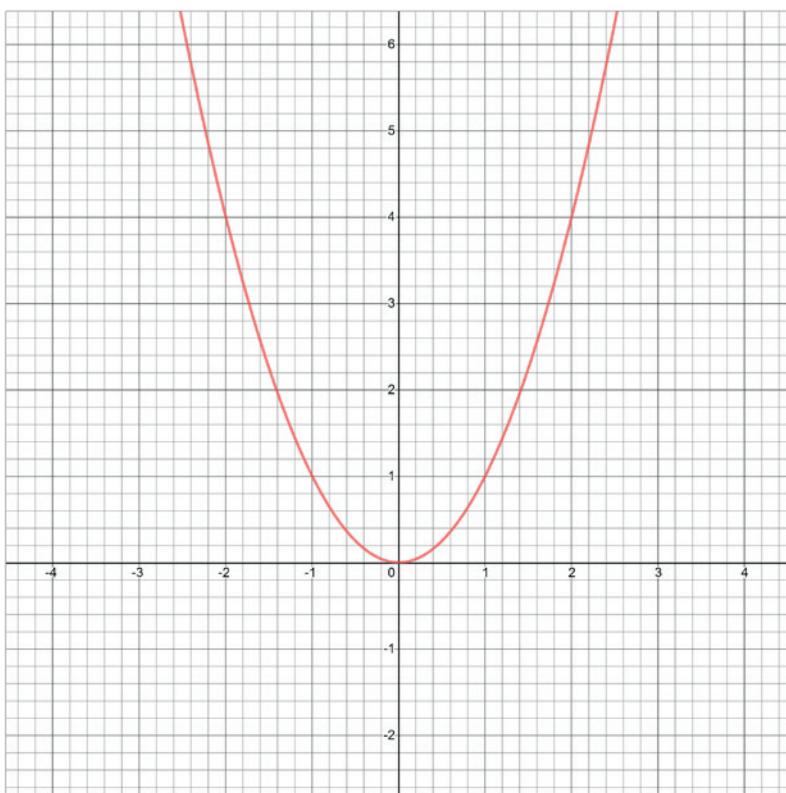
$$ECM = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m (SalidaRed_j^i - SalidaEsperada_j^i)^2$$

- **Cross-entropy** (CE). Esta función es utilizada en problemas de clasificación donde la salida esperada y la del modelo son distribuciones de probabilidad. Se calcula como el logaritmo natural de la probabilidad dada por la red para la clase positiva (la clase que tiene el valor 1 en el vector en codificación *one-hot* de la salida esperada). Si se dispone de un conjunto de  $n$  datos y  $m$  neuronas de salida, el CE se calcula como

$$CE = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m -\ln(SalidaRed_j^i) * SalidaEsperada_j^i$$

En la Figura 2.9 y 2.10 se muestran las gráficas de la función ECM y CE respectivamente.

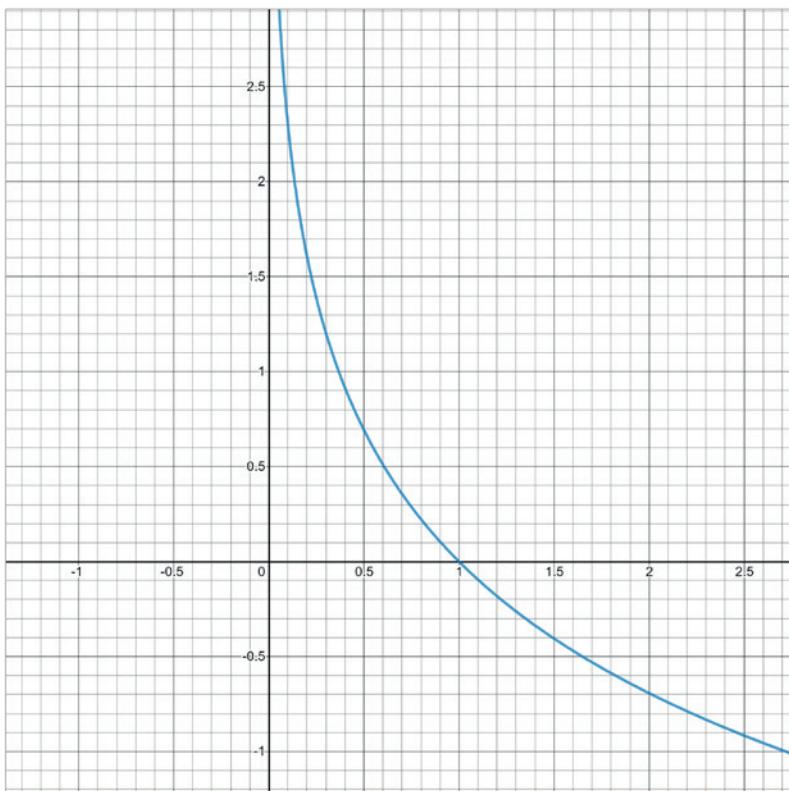
Tal y como se observa en la Figura 2.9, la función ECM comienza a dar valores altos a medida que la diferencia entre el valor esperado y el obtenido se aleja de la unidad. También se observa que el valor de salida es el mismo independientemente de si la diferencia entre el valor esperado y el obtenido es positiva o negativa; y es 0 en el caso de que el valor de salida coincida con el valor esperado. Por otro lado, como se aprecia en la Figura 2.10, la función CE podría dar como salida valores negativos. Sin embargo, la entrada de esta función está en el rango  $[0, 1]$  al tratarse de una probabilidad. Por lo tanto, siempre se va a obtener valores positivos de salida, concretamente, en el rango  $[0, \infty]$ . Cuando la salida de la red coincide con la salida esperada, el valor de esta función es 0, es decir, cuando el valor de salida de la red es 1 para la posición en que se espera que valga 1, el valor de la función CE es 0, lo que indica que la red no se ha desviado del valor esperado.



**Figura 2.9:** Gráfica de la función ECM

La justificación del uso de la función CE para problemas de clasificación basados en distribuciones de probabilidad, que se puede encontrar en la teoría de la información, queda fuera del alcance de este libro.

A diferencia de las métricas, las funciones de error o pérdida dan una medida directa del error o pérdida de la red respectivamente y permiten ajustar los pesos del modelo para que resuelvan un problema.



**Figura 2.10:** Gráfica de la función CE

## 2.6. Aprendizaje basado en el gradien- te

El problema de ajustar los pesos de una red neuronal se conoce como *aprendizaje o entrenamiento* de una red. Este problema se suele enfocar desde la perspectiva de un problema de optimización matemática.

Para el caso del aprendizaje supervisado, se dispone de un conjunto de datos donde,

para cada entrada,  $x$ , existe un valor de salida esperado,  $y$ . Este par de valores,  $(x, y)$ , provienen de cierta distribución  $\mathcal{D} : (x, y) \in \mathcal{D}$ . A esta tupla se hace referencia como ejemplo del conjunto de datos. En la práctica solo se cuenta con  $N$  muestras de ejemplos que conforman el conjunto de datos  $D : (x, y) \in D, D \subset \mathcal{D}$ . Esto se debe a que, en la práctica, es inviable obtener todos los posibles datos de un problema. Por ejemplo, en un problema de reconocimiento de vehículos en imágenes, es imposible tener un conjunto de datos con las imágenes de lo que se podría considerar un vehículo, aunque sí se puede conseguir una muestra de este conjunto.

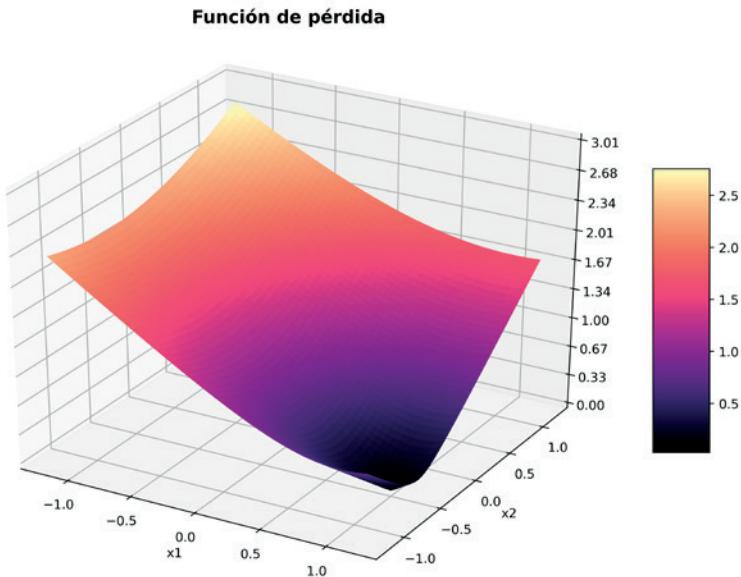
Además del conjunto de datos, para proceder al entrenamiento, se necesita una función de pérdida,  $\mathcal{L}$ , que estará parametrizada por los pesos de una red,  $W \in \mathbb{R}^N$ .

En base a todo lo anterior, el problema de optimización que se intenta resolver consiste en encontrar los valores de los pesos de la red tales que minimizan la función de pérdida, es decir,

$$W^* = \arg \min_W \mathcal{L}_W$$

Para comprender mejor en qué consiste este problema de optimización, es conveniente visualizar la superficie de la función de pérdida. Si se toman todos los datos del conjunto de entrenamiento y se calcula el valor de pérdida para cada posible combinación de valores, se puede dibujar la **superficie** de la función de pérdida. En la Figura 2.11 se muestra esta superficie para un perceptrón de ejemplo con dos entradas.

En el ejemplo de la Figura 2.11 la superficie es de tres dimensiones, puesto que el perceptrón tiene dos pesos. Si tuviese más pesos no se podría dibujar esta superficie de la función de pérdida. Se puede observar que, para este ejemplo, no existen valores de los pesos que hagan que el perceptrón no se equivoque. Siempre va a cometer algún error al clasificar todos los ejemplos. Esto puede ser debido a que, como ya se ha estudiado, existan *outliers* en los datos o que, directamente, no sean linealmente separables. No obstante, se puede deducir que los mejores valores para los pesos del perceptrón son  $x_1 = 1.0$  y  $x_2 = -0.25$ . Según se observa en la gráfica, no existen otros valores que vayan a conseguir reducir más el valor de

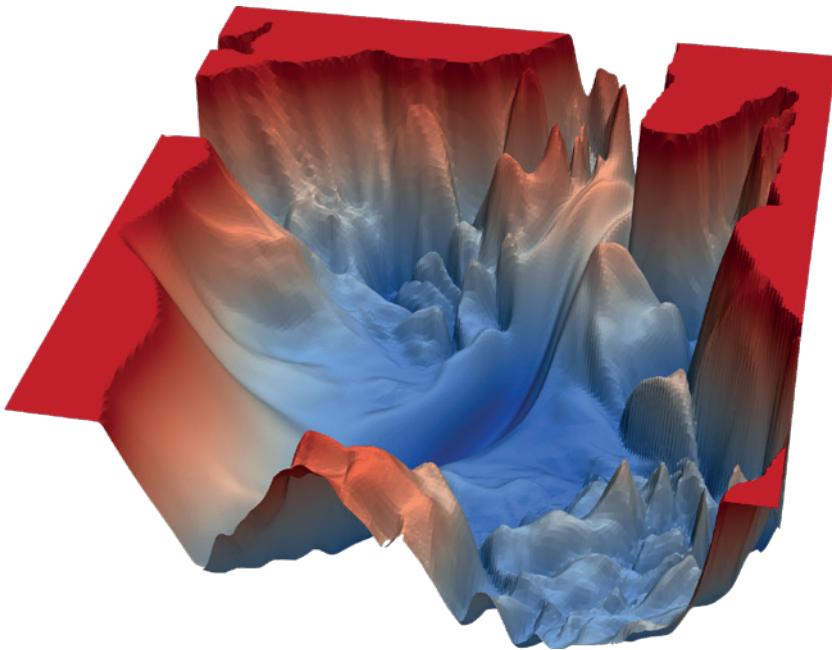


**Figura 2.11:** Superficie de la función de pérdida para un perceptrón con dos entradas

pérdida. Este punto es el mínimo global de la función de pérdida y es el que se intenta encontrar cuando se resuelve el problema de optimización.

Cuando se busca el mejor valor para los pesos de una red neuronal, la superficie de la Figura 2.11 no suele ser la habitual. Lo normal es que la superficie de la función de pérdida se parezca más a la mostrada en la Figura 2.12. Se trata de una superficie no convexa que cuenta con multitud de mínimos locales.

Un primer enfoque para abordar este problema de optimización podría ser el buscar una solución de forma analítica. Es sabido que se busca el mínimo de la función de pérdida, esto es, el punto en el que, para todos los pesos,  $w_i \in W$ , las derivadas son nulas,  $\frac{\partial \mathcal{L}_W}{\partial w_i} = 0$ . Sin embargo, la función de pérdida no es convexa y no existe una solución analítica para resolver  $\frac{\partial \mathcal{L}_W}{\partial w_i} = 0$ . Esta limitación se hace más evidente ante un problema de regresión. Por simplicidad, imagine que se utiliza un único perceptrón, sin función de activación y con la función ECM como función de



**Figura 2.12:** Superficie típica de la función de pérdida de una red neuronal

Fuente: <https://www.cs.umd.edu/~tomg/projects/landscapes/>

pérdida. En este caso, la ecuación que permite calcular los mejores valores para los parámetros de un modelo de regresión lineal (el perceptrón en este caso) es

$$W = (x^T x)^{-1} x^T y$$

Dado que se necesita hallar la inversa de  $x$  y esta puede no existir, no siempre se va a poder resolver esta ecuación. Además, el coste asociado a su cálculo es mayor que el de técnicas alternativas para encontrar este mínimo e incluso inviable de calcular debido a las dimensiones que puede llegar a tener esta matriz, que involucra a todos los datos del problema.

Otra posibilidad consiste en utilizar técnicas metaheurísticas como algoritmos evolutivos o una simple búsqueda aleatoria. Estas opciones quedan descartadas por la

elevada dimensionalidad del espacio de búsqueda. En la Figura 2.11 se mostraba una superficie en tres dimensiones donde se debe realizar la búsqueda. Los pesos de la red pueden ser cualquier valor real, por lo que la búsqueda, incluso en pocas dimensiones, es bastante compleja y difícil. Y aún lo es más cuando se trabaja con un problema real donde se puede llegar a tener una red con millones de parámetros que elevan la dimensión del espacio de búsqueda de manera exponencial.

Una posible alternativa al método analítico y al uso de técnicas metaheurísticas es el uso de métodos iterativos para resolver el problema de optimización. Con un método iterativo se puede ir calculando la solución al problema en pequeños pasos, lo que reduce las demandas computacionales comparado con el método analítico, y, aunque puede que no encuentre la solución óptima, siempre va a calcular una solución, lo que no estaba garantizado con el método analítico.

Además de las razones puramente técnicas para no usar los métodos analíticos, existe cierto interés en buscar métodos similares a los que utiliza el cerebro, al que intentan emular las redes neuronales. Es por ello que los primeros algoritmos para el entrenamiento de redes neuronales estaban inspirados en las teorías que se iban desarrollando sobre la forma en la que el cerebro aprende.

En 1949, Donald Hebb presentó una teoría de cómo el cerebro era capaz de aprender [7], muchas veces resumida con la frase «las neuronas que se disparan juntas permanecerán conectadas». Esta teoría viene a decir que si dos neuronas se mantienen activas al mismo tiempo, su conexión se fortalece. Frank Rosenblatt fue uno de los primeros en utilizar la teoría de Hebb para proponer un algoritmo de actualización de los pesos del perceptrón (ver Algoritmo 1).

Se observa que la forma de actualizar los pesos del algoritmo consiste en sumar el vector de pesos,  $w$ , con el vector de entrada,  $x$ , en el caso de que la entrada sea positiva ( $y = 1$ ) y el perceptrón la haya clasificado como negativa ( $w^T x < 0$ ); si la entrada es negativa ( $y = -1$ ) y el perceptrón la ha clasificado como positiva ( $w^T x > 0$ ) se le resta al vector de pesos el vector de entrada. La forma en la que actualiza los pesos del perceptrón se la conoce como **regla de actualización**. Rosenblatt demostró que el algoritmo converge en un tiempo finito si los datos son linealmente separables.

Una vez se le ha dado unos pesos iniciales al perceptrón, el algoritmo de entrenamiento solo modificará los pesos si encuentra algún ejemplo que el perceptrón

---

**Algoritmo 1** Algoritmo de entrenamiento del perceptrón

---

**Entrada:** El vector de pesos del perceptrón,  $w$ , y el conjunto de datos de entrenamiento,  $D : (x, y) \in D$  donde  $y \in \{-1, 1\}$

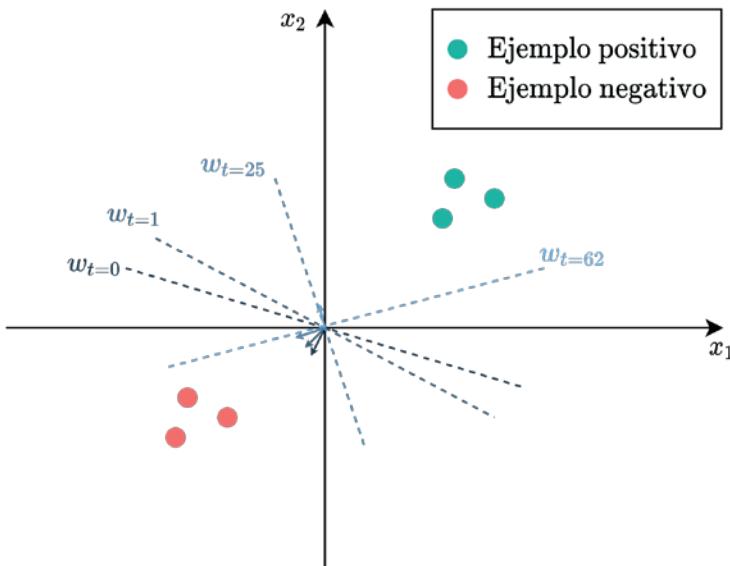
**Salida:** El vector de pesos  $w$  actualizado de forma que separa linealmente a los datos de  $D$

```
1: while !converge do
2:   Se toma un ejemplo  $(x, y) \in D$ 
3:   if  $y = 1$  y  $w^T x < 0$  then
4:      $w = w + x$ 
5:   end if
6:   if  $y = -1$  y  $w^T x > 0$  then
7:      $w = w - x$ 
8:   end if
9:   Cuando todos los ejemplos de  $D$  son clasificados correctamente el
    algoritmo converge
10:  end while
```

---

no sea capaz de clasificar correctamente. Cuando esto ocurre, para los ejemplos positivos, la suma con el vector de pesos provoca que el ángulo de separación entre estos se reduzca y, en el caso de los ejemplos negativos, aumente. En el momento que este ángulo permite que todos los ejemplos sean clasificados correctamente, el algoritmo converge. En la Figura 2.13 se ha representado, a modo de ejemplo, cómo podría ser la evolución de un vector de pesos,  $w$ , para un perceptrón con dos entradas a lo largo de diferentes iteraciones,  $t$ , del algoritmo de entrenamiento.

Con la llegada de ADALINE, el perceptrón pasó a ser únicamente la combinación lineal de las entradas. Dejando la función de activación aparte, ahora se podía aproximar funciones reales. Dado que el algoritmo original de Rosenblatt era incapaz de converger si los datos no eran linealmente separables, los autores pasaron a un enfoque más práctico y abordaron el problema desde una perspectiva puramente matemática de un problema de optimización. Definieron una función de pérdida (la función ECM fue la utilizada originalmente) y modificaron el algoritmo de entrenamiento del perceptrón para que los pesos de este fueran ajustados de forma proporcional al error que cometían. Para ello, modificaron la regla de actualización



**Figura 2.13:** Evolución del vector de pesos durante el algoritmo de entrenamiento del perceptrón

de los pesos y se introdujo una nueva regla, conocida como **regla delta**, que hacía uso del gradiente de la función de pérdida.

El gradiente de una función es un vector en el que se agrupan todas las derivadas parciales de todos los parámetros de la función. Para una función de pérdida,  $\mathcal{L}_W$ , parametrizada por los  $n$  pesos,  $W$ , de una red neuronal, el gradiente se define como

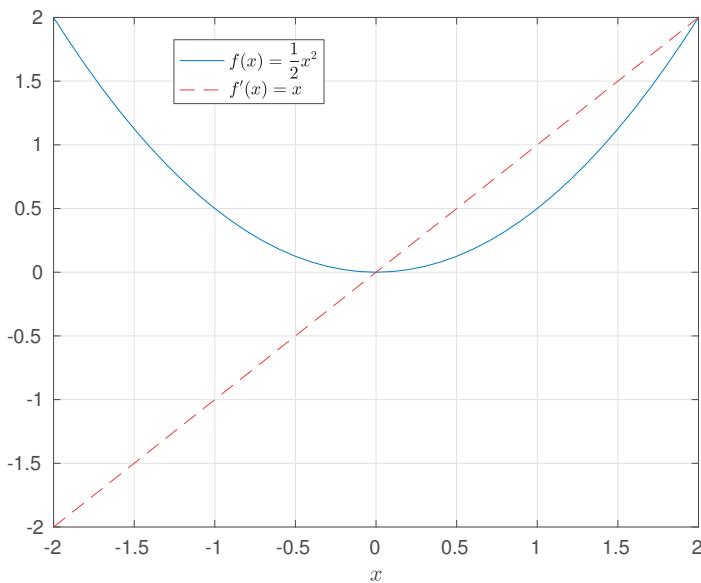
$$\nabla \mathcal{L}_W = \left[ \frac{\partial \mathcal{L}_W}{\partial w_1}, \dots, \frac{\partial \mathcal{L}_W}{\partial w_n} \right]$$

Para el caso particular de una función con un único parámetro,  $f(x)$ , se puede hablar directamente de la derivada de la función,  $f'(x)$ , en lugar del gradiente.

La utilidad del gradiente, o de la derivada, radica en que ofrece información del efecto que tiene, en el valor de salida de la función, el cambio de sus pesos en un sentido u otro. Esto proporciona información para encontrar el mínimo (ver Figura 2.14).

En base al valor de la derivada se sabe que:

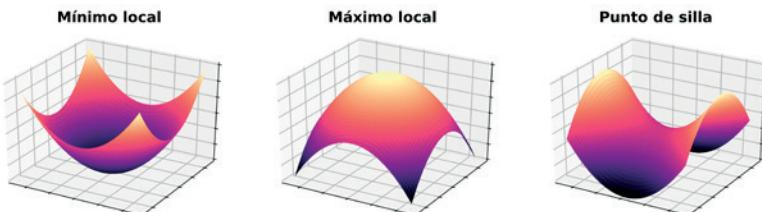
- Si  $f'(x) > 0 \implies$  hay que reducir  $x$  para acercarse al mínimo.
- Si  $f'(x) < 0 \implies$  hay que aumentar  $x$  para acercarse al mínimo.
- Si  $f'(x) = 0 \implies$  no aporta información de cómo modificar  $x$ , se trata de un óptimo local (máximo o mínimo).



**Figura 2.14:** Representación gráfica de una función y su derivada

Si la superficie de la función de pérdida se define en base a más de un parámetro, además de los casos anteriores, hay que añadir un nuevo caso, el conocido como punto de silla. Se trata de un punto crítico, que es un mínimo en algunas dimensio-

nes y un máximo en otras. En la Figura 2.15 se representan ejemplos de funciones con los puntos críticos que pueden aparecer.



**Figura 2.15:** Posibles puntos críticos de una función

La forma en la que se utiliza el gradiente en la regla delta, que no es más que un caso particular del algoritmo **descenso por gradiente**, se detalla, junto con otros algoritmos populares derivados a partir de este, en la Sección 2.6.2. Los principales problemas del entrenamiento basado en gradiente se analizan en la Sección 2.6.4.

La ventaja de los algoritmos de optimización basados en gradiente como la regla delta es que, al ser algoritmos iterativos, pueden abordar problemas que, de otra forma, serían intratables por los equipos de cómputo actuales. Otra ventaja es que, aunque los datos no fueran linealmente separables, se podría asignar a los pesos del perceptrón los valores que minimizan la función de pérdida. Aunque nunca vaya a valer cero, el algoritmo es capaz de converger al mínimo de la función, lo que no estaba garantizado en el algoritmo de entrenamiento original.

Independientemente del algoritmo de optimización escogido, se suele distinguir tres tipos de optimización en base al número de ejemplos del conjunto de datos,  $D : (x, y) \in D$ , que se utilicen durante el cálculo del gradiente:

- **Optimización por lotes** (*batch* en inglés). En este caso se hace uso de todos los ejemplos disponibles en cada paso. El principal problema de esta estrategia es que, en muchos casos, es inviable, debido a la cantidad de ejemplos disponibles. Ralentiza demasiado el entrenamiento o lo hace imposible por las limitaciones de memoria para almacenarlos en un mismo instante. El problema se plantea como:

$$\arg \min_W \mathcal{L}_{W,D}$$

- **Optimización estocástica** (*stochastic* en inglés). Es el caso opuesto al anterior. Solo se hace uso de un único ejemplo para actualizar el modelo en cada iteración. Esta forma de entrenar no se suele utilizar debido a la dificultad de conseguir un buen rendimiento general del modelo utilizando un único ejemplo. El planteamiento del problema sería:

$$\arg \min_W \mathcal{L}_{W,(x^{(i)},y^{(i)})}$$

- **Optimización por mini-lotes** (*mini-batch* en inglés). Esta estrategia presenta un compromiso entre un modelo con un buen rendimiento general y la velocidad del entrenamiento. Se utilizan  $n$  ejemplos para optimizar el modelo. Esta estrategia es la más utilizada. El problema de optimización es:

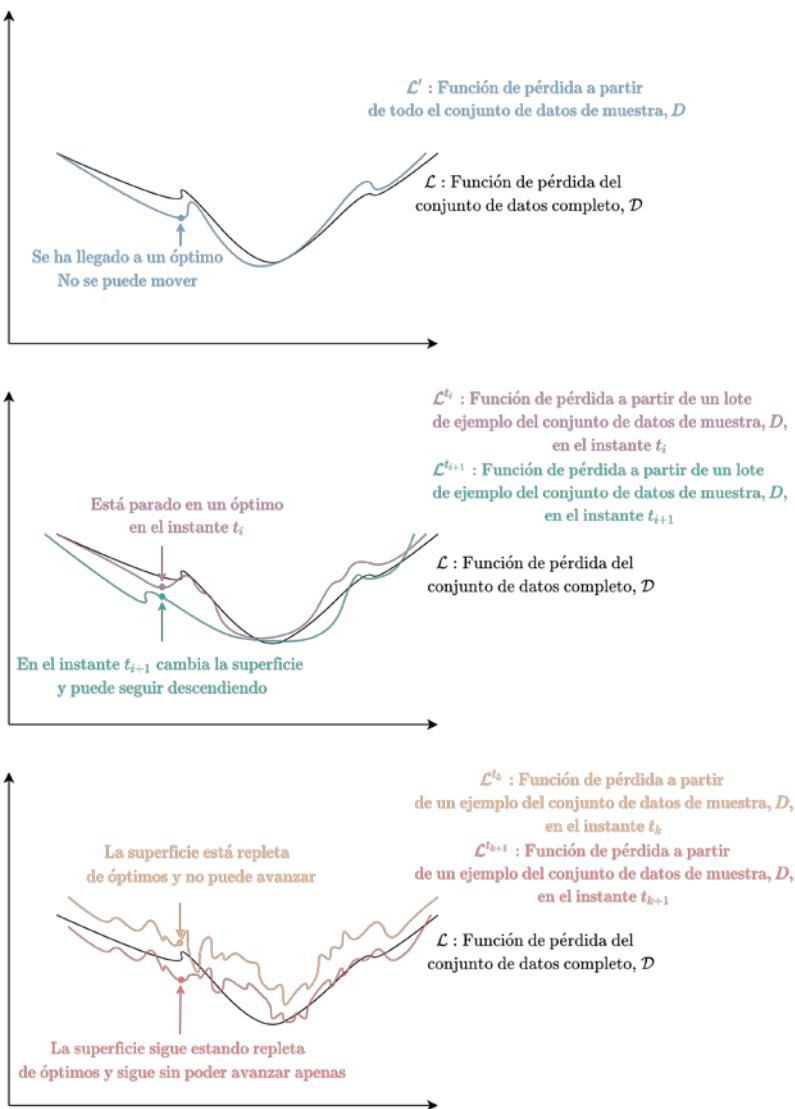
$$\arg \min_W \mathcal{L}_{W,(x^{(i:i+n)},y^{(i:i+n)})}$$

Lo habitual es utilizar la optimización por mini-lotes donde el tamaño del lote, o *batch size* en inglés, se define de forma que el equipo a utilizar pueda alojar en memoria los ejemplos para calcular el gradiente. El hecho de que esta optimización sea la más popular tiene que ver con el efecto que produce en la superficie de la función de pérdida utilizar más o menos ejemplos. Si se utilizan todos los ejemplos disponibles, además de ser inviable por la imposibilidad de almacenar todos los datos en memoria, la superficie de la función de pérdida es siempre la misma en todas las iteraciones del algoritmo. Cuando se caiga en un mínimo local no se podrá salir de él. En el caso de la optimización por mini-lotes o estocástica, en cada iteración la superficie de la función de pérdida va cambiando, ya que se define en base al error que comete la red al clasificar los datos de ejemplo, que no son siempre los mismos. Si los datos de la iteración actual son diferentes a los de la anterior, seguramente, la superficie varíe lo suficiente como para que el mínimo local en el que esté detenido deje de serlo y se pueda escapar de él. Sin embargo, cuanto menor es el número de ejemplos para calcular el gradiente, menor es la probabilidad que el mínimo de la función de pérdida coincida con el mínimo de la función real, por lo que se intenta evitar la optimización estocástica siempre que es posible.

El mínimo más parecido al real debe ser el que definen todos los ejemplos de entrenamiento, pero cuenta con el problema de que no se puede calcular el gradiente tomando todos los datos y que el algoritmo de optimización puede quedar atrapado en algún mínimo local. Por lo tanto, la optimización por mini-lotes es la que ofrece un mejor compromiso entre utilizar un buen mínimo de referencia y poder evitar mínimos locales. En la Figura 2.16 se comparan las superficies de la función de pérdida que definirían cada una de las estrategias de optimización para un problema de ejemplo y el efecto que tiene en el entrenamiento cada una.

Cuando Minsky y Papert estudiaron el perceptrón, quedó demostrada la necesidad de utilizar redes neuronales con, al menos, una capa oculta y función de activación no lineal, para tratar problemas no separables linealmente. El problema de las redes neuronales con capas ocultas es que no se conoce la salida esperada en las neuronas de estas capas y, por lo tanto, no se puede utilizar un algoritmo de entrenamiento basado en el gradiente de la función de pérdida porque no se conoce el error que cometen estas neuronas. Esta limitación fue la que provocó el primer invierno de la inteligencia artificial y el abandono de las redes neuronales. No fue hasta la llegada del algoritmo *backpropagation*, en la década de los 80, que se encontró la manera de ajustar los pesos de las capas internas, lo que propició que se recuperara el interés por las redes neuronales. El estudio de este algoritmo se aborda en la Sección 2.6.3.

Antes de acabar esta sección cabría plantearse si el uso del gradiente es la mejor forma que existe para definir los pesos de una red. Las técnicas basadas en el gradiente hacen uso de la primera derivada, por lo que no obtienen información de la curvatura de la función de pérdida, solo se puede saber cómo de rápido baja la pérdida, pero no si la curva es plana, curvada hacia arriba o hacia abajo. Existen alternativas como el método de Newton que, en lugar de utilizar la primera derivada, usa la segunda para alcanzar el punto donde está el mínimo. Aunque este método pueda llegar al mínimo de forma más rápida, el problema de este y otros métodos basados en la segunda derivada es su elevado coste computacional, tanto de procesamiento como de almacenamiento. Además, al basarse en el cálculo de las raíces de la derivada, puede darse el caso de que no exista o que sí exista, pero nunca sepamos si es un máximo o un mínimo. Estos métodos se acercan a un óptimo que puede ser de cualquier tipo: puede acabar en un máximo cuando el objetivo sea encontrar un mínimo y viceversa. Hasta la fecha, los algoritmos de optimización iterativos basados en el gradiente son los más utilizados.



**Figura 2.16:** Efecto en la definición de la superficie de error de las diferentes estrategias de optimización

## 2.6.1. Inicialización de los pesos

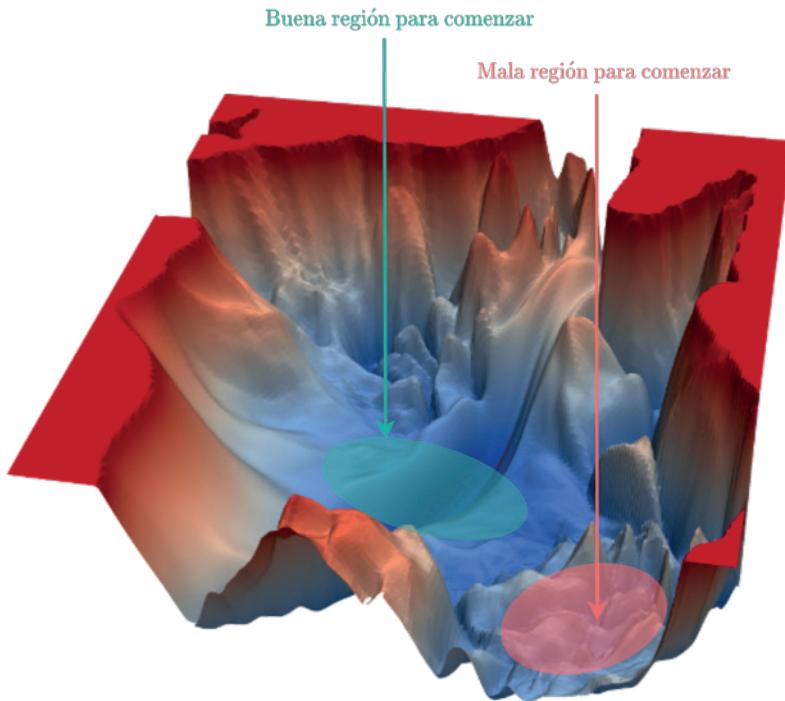
Todos los algoritmos de entrenamiento van a necesitar de un punto de partida desde el que ir modificando los pesos de la red. La inicialización de los pesos marca en gran medida los resultados que se podrán alcanzar.

Hay que tener presente que, según se inicialicen los pesos de la red, estos se situarán en un punto determinado de la superficie de la función de pérdida. El objetivo es que esta inicialización los sitúe cerca del mínimo global de la función. En la Figura 2.17 se indican las regiones de la superficie de error en las que interesa comenzar el entrenamiento (regiones próximas al mínimo global) y las que no (alejadas del mínimo global y repleta de mínimos locales en los que el algoritmo podría detenerse).

A priori, no se conoce la superficie de la función de pérdida, por lo que no hay indicios de qué valores son mejores que otros para inicializar los pesos. La solución más simple sería hacer una inicialización de los pesos puramente aleatoria. Sin embargo, dada la alta dimensionalidad de la función de pérdida, es casi imposible que acaben en una región próxima al mínimo mediante una inicialización aleatoria.

Otra alternativa podría ser inicializar todos los pesos a cero o a algún otro valor. El principal inconveniente de esta técnica es que, al depender del gradiente del error para actualizar los pesos y tener todos el mismo valor, los pesos se actualizan de la misma forma, quedando todos con el mismo nuevo valor resultado de la actualización. Si la solución al problema requiere que los pesos de las neuronas sean diferentes, no se podrá conseguir utilizando esta inicialización. Es necesario que los pesos sean inicializados con valores diferentes para romper esta simetría.

Además de tener que romper la simetría, no se pueden inicializar los pesos de la red de cualquier forma. Si, por ejemplo, se utilizan unos valores muy pequeños y la red tiene muchas capas, se podría acabar generando valores muy próximos o iguales a cero en las capas finales de la red. La sucesiva multiplicación de las entradas por valores muy grandes harían que estas se desvanecieran. Por otro lado, puede ocurrir lo contrario, es decir, que al inicializar los pesos con unos valores muy altos se generen valores de salida demasiado grandes, haciendo que, incluso entradas muy similares, tengan una salida muy diferente, lo que dificulta en gran medida el entre-



**Figura 2.17:** Regiones de la superficie del error ideales para comenzar el entrenamiento

namiento, pudiendo llegar a producirse errores por desbordamientos al superarse el valor máximo del tipo de dato con el que se almacenan los pesos.

Para evitar todos los problemas anteriores, se desarrollaron nuevas técnicas de inicialización aleatoria más elaboradas que, a continuación, se detallan:

- **Inicialización Xavier.** Esta técnica surge como solución al problema que aparece cuando se hace una inicialización puramente aleatoria [8]. Según la función de activación que se utilice, se considera que se satura al llegar al valor máximo o mínimo que da como salida. Si se inicializan los pesos de forma aleatoria, puede que se provoque esta situación y muchas salidas estén saturadas; lo que exige más iteraciones al algoritmo de entrenamiento para ajustar los pesos. Sus

autores sugieren inicializar los pesos de forma que la varianza de los pesos de una neurona sea igual a la unidad. De esta forma, se reduce la probabilidad de situarlos sobre las zonas que provocan la saturación de la función de activación. Considerando una neurona,  $j$ , con  $n$  pesos, la varianza de la neurona será:

$$\begin{aligned} \text{Var(neurona}^j) &= \text{Var}(w_1^j x_1 + \cdots + w_n^j x_n) = \\ &= n \text{Var}(w_i^j) \quad \forall i \in [1, \dots, n] \end{aligned}$$

Por tanto, esta inicialización consiste en dar un valor a los pesos de la neurona a partir de una distribución normal:

$$w_i^j \sim N\left(0, \sqrt{\frac{1}{n}}\right)$$

- **Inicialización He.** Según sus autores, esta técnica es la mejor alternativa para inicializar una red que utilice funciones de activación ReLU [9]. Permite la convergencia de modelos extremadamente profundos (con muchas capas), para los cuales la inicialización Xavier no era eficaz. Consiste en asignar un valor aleatorio tomado de una distribución uniforme:

$$w_i^j \sim U\left(0, \sqrt{\frac{2}{n}}\right)$$

donde  $n$  es el número de entradas de la neurona  $j$ .

El *bias* de las neuronas se suele inicializar a cero. En este caso no hay riesgo de que haya simetría, ya que esta se rompe con cualquiera de las inicializaciones aleatorias anteriores para el resto de los pesos de la neurona.

Existe una alternativa a todo lo anterior que consiste en partir de una red ya entrenada. Imagine que ha utilizado una red, con buenos resultados, para un problema bastante complejo y genérico, o que algún otro investigador lo ha hecho y ha facilitado una red con los pesos ya ajustados. En lugar de empezar un entrenamiento desde cero para un nuevo problema con alguna inicialización aleatoria, se podría utilizar una red que ya haya sido entrenada y así aprovechar lo que esta red haya sido

capaz de aprender para acelerar el entrenamiento. Esta técnica de inicialización se conoce como transferencia de aprendizaje o *transfer learning* en inglés. Se transfiere lo aprendido en un problema a uno nuevo, en principio más sencillo que el original, con la idea de comenzar desde una zona más próxima al mínimo que con la inicialización aleatoria, gracias a lo ya aprendido la red en el otro problema.

## 2.6.2. Principales algoritmos de optimización

En esta sección se estudian algunos de los algoritmos de optimización basados en el uso del gradiente más utilizados para el entrenamiento de redes neuronales. Todos parten de la idea básica del algoritmo de descenso por gradiente y le aportan ciertas mejoras para conseguir una convergencia más rápida y alcanzar un mejor mínimo.

A diferencia del primer algoritmo de entrenamiento del perceptrón, estos algoritmos no tienen un criterio de convergencia, sino que se define un número de iteraciones durante las que se modifican los pesos en base a la regla de actualización específica de cada uno. En Algoritmo 2 se detalla la estructura común de todos estos algoritmos basados en el gradiente para el caso de la optimización por mini-lotes.

---

### **Algoritmo 2** Algoritmos basados en el gradiente

---

**Entrada:** La matriz de pesos de la red,  $W$ , el conjunto de datos de entrenamiento,  $D : (x, y) \in D$  y la función de pérdida,  $\mathcal{L}$

**Salida:** La matriz de pesos  $W$  al finalizar el entrenamiento

- 1: Se inicializa  $W$
  - 2: **for**  $k = 1$ :numIteraciones **do**
  - 3:     Se toma un mini-lote de ejemplos,  $(x^{(i;i+n)}, y^{(i;i+n)}) \in D$
  - 4:      $W_{k+1} = \text{reglaDeActualizacion}(\mathcal{L}_{W_k, x^{(i;i+n)}, y^{(i;i+n)}})$
  - 5: **end for**
- 

Es importante recordar que no se dispone de la salida esperada para las capas ocultas

de la red por lo que, hasta que no se presente el algoritmo *backpropagation*, no se podrá utilizar estos algoritmos en redes con más de una capa.

### 2.6.2.1. Algoritmo de descenso por gradiente

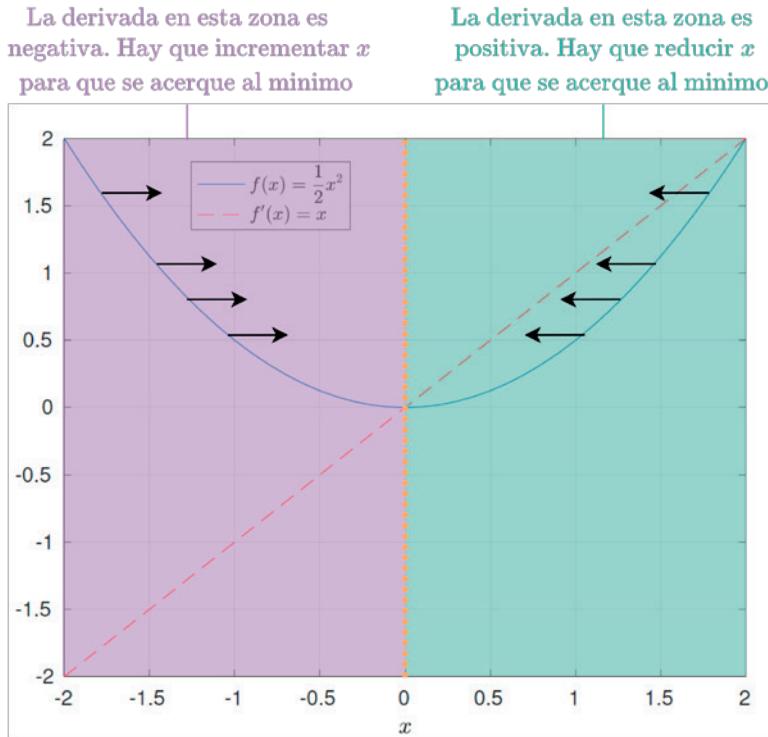
El algoritmo de descenso por gradiente se atribuye al matemático francés, Augustin Louis Cauchy, quien lo presentó en el año 1847. Este algoritmo hace uso del gradiente para actualizar, en cada paso, los valores de los pesos de la red en el sentido contrario al signo de la derivada. Como se vio en la Sección 2.6, esto permite acercarse al mínimo (ver Figura 2.18). De esta forma, mediante un proceso iterativo, los pesos acaban siendo los que se sitúan en el mínimo de la función de pérdida que indica el gradiente. Parte de un punto inicial en la superficie de la función de pérdida, que viene dado por la inicialización realizada de los pesos y, desde este punto, se van actualizando los pesos,  $W$ , una cierta cantidad en cada paso,  $t$ , como indica la regla de actualización de este algoritmo:

$$W_{t+1} = W_t - \nabla_W \mathcal{L}_{W_t}$$

Para el caso de la función de pérdida ECM, la regla de actualización es conocida como la regla delta, que fue introducida por los autores de ADALINE como ya se ha estudiado.

El problema de utilizar directamente el gradiente es que no siempre se podrá alcanzar el mínimo. La cantidad que se modifica los pesos en cada iteración es conocida como paso y viene marcado por el valor del gradiente. Si están lejos del mínimo, un valor pequeño de gradiente puede que no permita alcanzarlo al finalizar el entrenamiento. Por otro lado, si están cerca y el valor es alto comenzarán a pasar de un lado a otro por encima del mínimo sin llegar a bajar. Este y otros problemas del entrenamiento basado en gradiente se analizan en detalle en la Sección 2.6.4.

Para controlar el efecto del gradiente en la actualización de los pesos se introduce un parámetro,  $\alpha$ , conocido como tasa de aprendizaje, o *learning rate* en inglés, que permite controlar el paso que da el algoritmo de entrenamiento en cada actualiza-



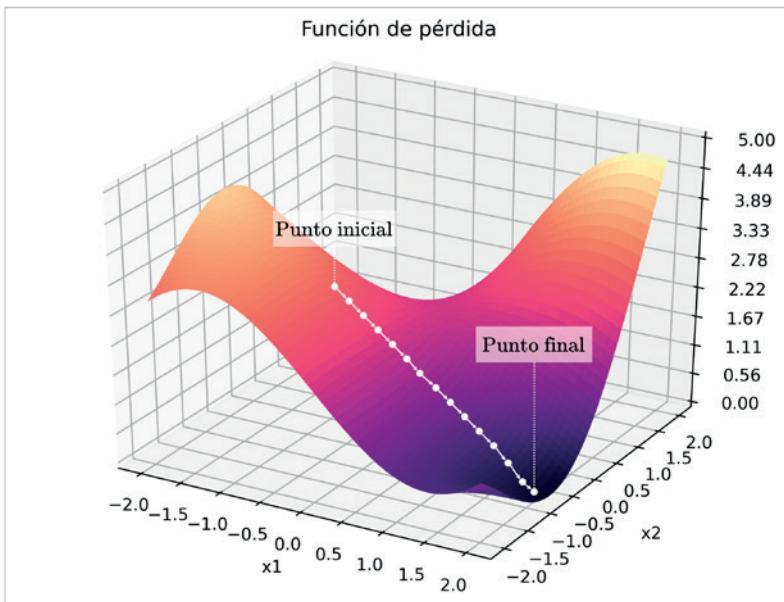
**Figura 2.18:** Modificación de la variable para acercarse al mínimo de la función

ción de los pesos. La regla de actualización de los pesos con este parámetro quedaría como

$$W_{t+1} = W_t - \alpha \nabla_W \mathcal{L}_{W_t}$$

Como se estudiará en la Sección 2.8, la elección de un buen *learning rate* es esencial para lograr la convergencia al mínimo.

En la Figura 2.19 se muestra un ejemplo de la modificación de los pesos realizada por este algoritmo sobre la superficie de la función de pérdida.



**Figura 2.19:** Modificación de los pesos por el algoritmo descenso por gradiente

Aunque efectivo, este algoritmo tiene el problema de que necesita muchas iteraciones para llegar al mínimo. En lugar de actualizar los pesos en base al gradiente actual se puede tener en cuenta el gradiente en iteraciones anteriores. Imagine que, durante un cierto número de iteraciones, el gradiente viene apuntando en la misma dirección. En lugar de ir actualizando en cada iteración un paso determinado, se puede dar un paso más grande, ya que todo parece indicar que es la dirección que se debe seguir. El algoritmo lleva un tiempo pidiendo seguir esa dirección así que no hay que perder más tiempo. Esta es la idea detrás del momento, o *momentum* en inglés. Es decir, en lugar de utilizar el gradiente actual, usar la media móvil exponencial de los gradientes anteriores, para lo que añade un parámetro,  $\beta \in [0, 1]$ , a la regla de actualización, que controla cuánto del gradiente actual y cuánto de

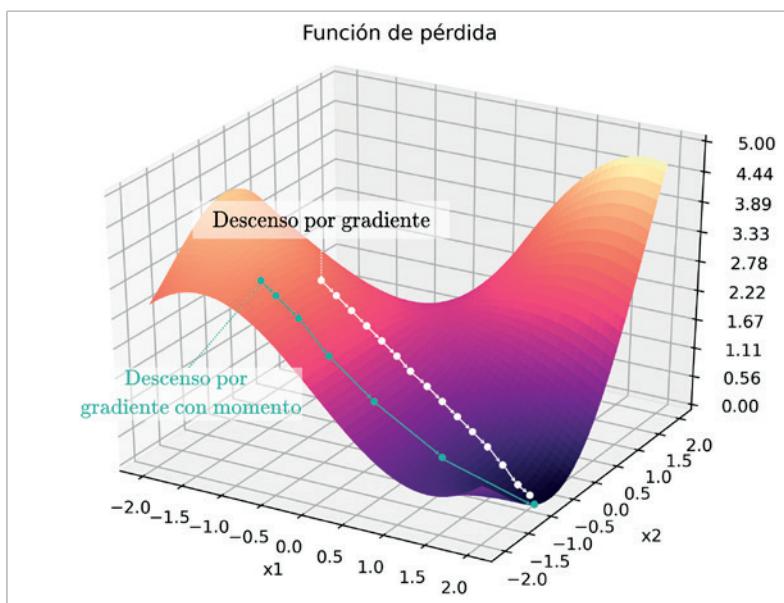
los anteriores es utilizado. Se puede imaginar que los pesos se van moviendo como una bola en la superficie de la función de pérdida  $y$ , conforme bajan en la superficie, van acumulando momento que les hace descender más rápido. La regla de actualización para el descenso por gradiente con momento es

$$v_t = \beta v_{t-1} + \alpha \nabla_W \mathcal{L}_{W_t}$$

$$W_{t+1} = W_t - v_t$$

Normalmente, se suele usar  $\beta = 0.9$  o un valor similar.

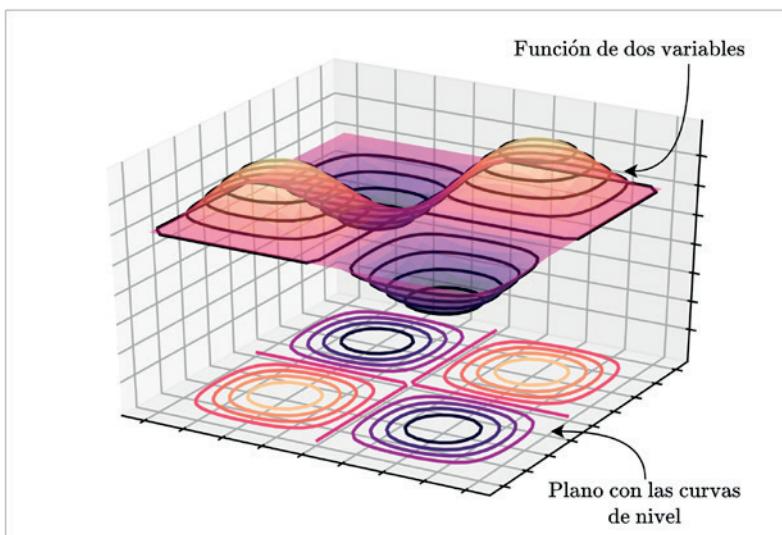
En la Figura 2.20 se muestra una comparativa entre el algoritmo de descenso por gradiente con y sin momento. Se observa que, cuando se utiliza el momento, se llega al mínimo en menos iteraciones.



**Figura 2.20:** Comparativa del algoritmo de descenso por gradiente con y sin momento

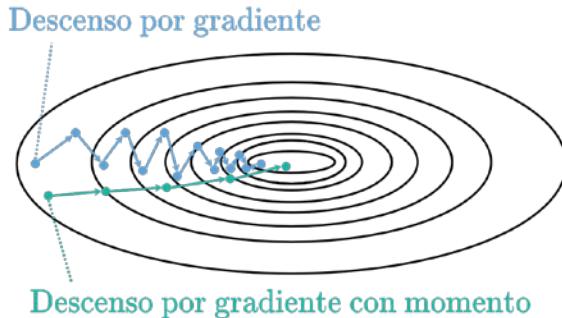
Otro problema que corrige el momento es el descenso en zonas donde la curva de la superficie es mucho más abrupta en una dimensión que en otra. En estos casos, el descenso por gradiente sin momento oscila a través de la zona estrecha, ya que el gradiente apuntará hacia uno de los lados más que al otro a lo largo de la superficie.

Para entender mejor este problema se puede representar la superficie de la función de pérdida en el plano mediante curvas de nivel. Las curvas de nivel para una función de dos variables, como las utilizadas para los ejemplos de la función de pérdida, son una familia de curvas que representan todos los puntos que tienen la misma imagen, es decir, la misma altura. En la Figura 2.21 se muestra una función de dos variables y sus curvas de nivel.



**Figura 2.21:** Función de dos variables y plano con sus curvas de nivel

Cuando usamos el momento, el gradiente aumenta en las dimensiones cuyos gradientes apuntan en la misma dirección, produciendo una convergencia más rápida y una reducción de la oscilación (ver Figura 2.22).



**Figura 2.22:** Reducción de la oscilación por el uso del momento

En el Ejemplo 2.1 se muestra un ejemplo del funcionamiento del algoritmo descenso por gradiente.

**Ejemplo 2.1 :** Interesa encontrar el mínimo de una función,  $f$ , de una variable, definida como  $f(x) = \frac{1}{2}x^2$ . El algoritmo de descenso por gradiente utiliza la regla de actualización

$$x_{t+1} = x_t - \alpha \nabla_x f(x_t)$$

en la que se establece un valor de  $\alpha = 0.5$  para el *learning rate*. El gradiente de la función es directamente la derivada respecto de  $x$ , ya que solo tiene una variable, y este es  $f'(x) = x$ , por lo que la regla de actualización queda como

$$x_{t+1} = x_t - 0.5x_t$$

Para poder aplicar el algoritmo descenso por gradiente hay que inicializar la variable, por ejemplo,  $x = 3$ . En base a todo lo anterior, las cinco primeras iteraciones del algoritmo transcurrirían como se muestra a continuación:

- **Iteración 1.** Desde donde se encuentra la variable,  $x_1 = 3$ , se calcula el valor para la siguiente iteración aplicando la regla de actualización, es decir,

$$\begin{aligned}x_2 &= x_1 - 0.5x_1 \\x_2 &= 3 - 0.5 * 3 = 1.5\end{aligned}$$

- **Iteración 2.** Se vuelve a aplicar la regla de actualización:

$$\begin{aligned}x_3 &= x_2 - 0.5x_2 \\x_3 &= 1.5 - 0.5 * 1.5 = 0.75\end{aligned}$$

- **Iteración 3.** Se vuelve a aplicar la regla de actualización:

$$\begin{aligned}x_4 &= x_3 - 0.5x_3 \\x_4 &= 0.75 - 0.5 * 0.75 = 0.375\end{aligned}$$

- **Iteración 4.** Se vuelve a aplicar la regla de actualización:

$$\begin{aligned}x_5 &= x_4 - 0.5x_4 \\x_5 &= 0.375 - 0.5 * 0.375 = 0.1875\end{aligned}$$

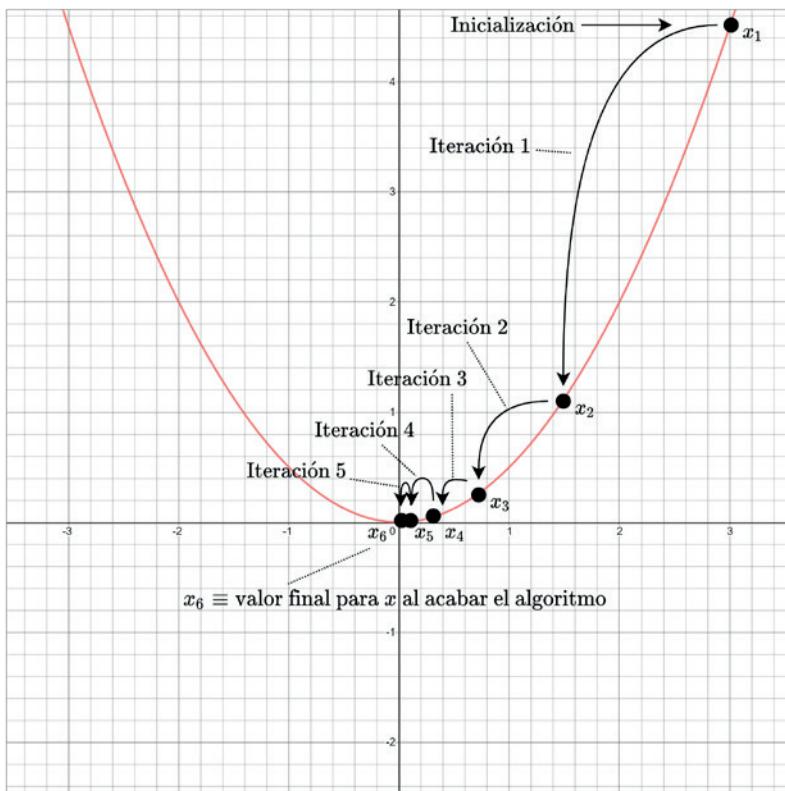
- **Iteración 5.** Se vuelve a aplicar la regla de actualización:

$$\begin{aligned}x_6 &= x_5 - 0.5x_5 \\x_6 &= 0.1875 - 0.5 * 0.1875 = 0.09375\end{aligned}$$

En la Figura 2.23 se muestra el avance del algoritmo desde la inicialización hasta la última iteración.

En el ejemplo anterior, la función de pérdida se define a partir del valor de  $x$ . Cuando se entrena una red neuronal por mini-lotes, la función de pérdida queda definida en base a la salida esperada y la salida dada por la red para la entrada del lote de datos de ejemplo actual.

La salida de la red depende tanto de los pesos como de las entradas. Si se mantienen



**Figura 2.23:** Ejemplo de iteraciones del algoritmo descenso por gradiente

los pesos, pero cambian las entradas, el valor de pérdida cambiara también. Lo mismo ocurre si se cambian los pesos y se mantienen las entradas. Por lo que, en cada iteración, como ya se ha estudiado, la función de pérdida va cambiando y no tiene siempre la misma forma.

### 2.6.2.2. Adagrad

Este algoritmo busca mejorar el descenso por gradiente haciendo uso de un *learning rate* adaptativo para cada peso de la red [10]. En lugar de utilizar el mismo valor para todos los pesos, los autores de Adagrad proponen que cada peso utilice el gradiente acumulado para actualizar su *learning rate*. De esta forma, se consigue un *learning rate* más alto para los pesos que no oscilan y un valor bajo para los pesos que oscilan. Esto permite que el proceso de optimización se centre en utilizar los pesos que le son realmente útiles e ignorar a aquellos que no aportan información segura del gradiente. La regla de actualización de este algoritmo divide el *learning rate* por la raíz cuadrada de la suma acumulada de los cuadrados de los gradientes de iteraciones anteriores como se muestra a continuación:

$$\begin{aligned} S_t &= S_{t-1} + [\nabla_W \mathcal{L}_{W_t}]^2 \\ W_{t+1} &= W_t - \frac{\alpha}{\sqrt{S_t + \epsilon}} \nabla_W \mathcal{L}_{W_t} \end{aligned}$$

$S$  se suele inicializar a cero y  $\epsilon$  es un valor pequeño que evita la división por cero.

Una de las ventajas de esta técnica es que elimina la necesidad de ajustar manualmente el *learning rate* con la contrapartida de que la acumulación del gradiente puede provocar que el entrenamiento acabe deteniéndose al tener un *learning rate* demasiado pequeño.

### 2.6.2.3. RMSprop

Con el objetivo de corregir el problema de Adagrad, RMSprop adapta el *learning rate* de cada peso usando la media móvil exponencial del gradiente. A diferencia de la simple acumulación de gradientes de Adagrad, la media móvil exponencial permite que los gradientes más antiguos acaben «olvidándose», previniendo de una parada prematura del entrenamiento. Este algoritmo no ha sido publicado y se dio a conocer a través de los apuntes de unas clases en línea del profesor Geoffrey Hinton. Su regla de actualización es la siguiente:

$$v_t = \rho v_{t-1} + (1 - \rho) [\nabla_W \mathcal{L}_{W_t}]^2$$

$$W_{t+1} = W_t - \frac{\alpha}{v_t + \epsilon} \nabla_W \mathcal{L}_{W_t}$$

donde  $\rho$  es la tasa de caída o *decay rate* (normalmente 0.9) y  $\epsilon$  representa un valor muy pequeño para evitar la división por cero.

### 2.6.2.4. Adam

*Adaptive Moment Estimation* o Adam [11] es un algoritmo que busca combinar las mejoras que aporta el uso del momento y un *learning rate* adaptativo. Adam calcula un *learning rate* adaptativo utilizando la media móvil exponencial del gradiente (como en el algoritmo RMSprop) además de usar la media móvil exponencial del gradiente en lugar del gradiente de la iteración actual (como el descenso por gradiente con momento).

Adam utiliza unos estimadores para calcular los momentos, pero, debido a que estos estimadores son inicializados a cero, los autores observaron que estaban sesgados hacia este, especialmente durante las primeras iteraciones. Como medida para contrarrestar este efecto calcularon los estimadores de forma que se corrigiera este sesgo:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_W \mathcal{L}_{W_t}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) [\nabla_W \mathcal{L}_{W_t}]^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

La regla de actualización de parámetros es la siguiente:

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Los autores proponen como valores por defecto 0.9 para  $\beta_1$  y 0.999 para  $\beta_2$ .  $\epsilon$  representa un valor muy pequeño para evitar la división por cero.

### 2.6.3. *Backpropagation*

A partir del estudio de las limitaciones del perceptrón realizado por Minsky y Papert, las redes neuronales cayeron en el olvido. No fue hasta la década de los 80 cuando estas volvieron a ganar protagonismo gracias al trabajo de David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams [12] donde se mostraba por primera vez, de forma práctica, cómo utilizar el algoritmo *backpropagation*, o propagación hacia atrás en español, para entrenar redes neuronales con más de una capa. Originalmente, el algoritmo *backpropagation* se desarrolló durante la década de los 60 y 70 por investigadores independientes, aunque no fue hasta su demostración práctica cuando este empezó a ser utilizado para entrenar redes neuronales.

La idea que subyace en este algoritmo es el uso de la regla de la cadena, utilizada para el cálculo de la derivada de una composición de funciones (básicamente lo que es una red neuronal), para el cálculo del gradiente de la función de pérdida en función de los pesos de las capas ocultas. Aunque no se conozca el error de los pesos de las capas ocultas porque no se sabe la salida esperada, sí se puede saber cómo afecta cada peso al error que comete la red a la salida, puesto que se calcula a partir de la composición de las operaciones de cada capa. Por tanto, ahora es posible calcular el gradiente de la función de pérdida, no solo para los pesos de las neuronas de la capa de salida, sino para todos los pesos de la red. *Backpropagation* ofrece una solución al problema del cálculo del gradiente en las capas ocultas pero no cómo utilizarlo. Las formas de utilizar este gradiente para actualizar los pesos siguen siendo las propuestas por los algoritmos de la Sección 2.6.2 solo que, con *backpropagation*, es posible aplicarlos en redes con más de una capa.

En el caso de una red neuronal, el objetivo del algoritmo de entrenamiento, es minimizar la función de pérdida,  $\mathcal{L}$ , a partir de pares de ejemplos,  $(x, y)$ , del conjunto de entrenamiento. En concreto, dado un dato de entrada,  $x$ , el valor de la función de pérdida,  $\mathcal{L}_W(\hat{y}(x), y)$ , se calcula en base a la distancia entre la salida esperada,  $y$ , la salida obtenida por la red,  $\hat{y}(x)$ , y los pesos de esta,  $W$ . La salida de

la red se define (ver Ecuación 1.6) mediante una composición de funciones en la que intervienen los pesos de la red.

En base a lo anterior, para minimizar la función de pérdida, hay que calcular la derivada respecto a todos y cada uno de los pesos de la red. Para lograrlo, se comienza derivando la función de pérdida para luego ir aplicando la regla de la cadena (ver Observación 2.1) a través de la composición de funciones que ha generado la salida de la red (funciones de activación incluidas) hasta llegar al peso que se esté ajustando.

**Observación 2.1 :** Según la regla de la cadena, la derivada de una función compuesta

$$z = (g \circ f)(x) = g(f(x))$$

se calcula como

$$z' = (g \circ f)'(x) = g'(f(x))f'(x)$$

o, con la notación de Leibniz,

$$z' = \frac{dz}{dx} = \frac{dz}{df} \frac{df}{dx}$$

El algoritmo *backpropagation* cuenta con dos etapas. En la primera de ellas, conocida como *forward pass* en inglés, se obtiene el valor de pérdida junto con el resultado de todas las operaciones intermedias de la red. En la segunda etapa, *backward pass* en inglés, al tener el valor de pérdida obtenido en el *forward pass* y los resultados de las operaciones intermedias, se calcula el gradiente de la función de pérdida.

Durante el *backward pass*, los primeros pesos sobre los que se calculará la derivada son los de la capa de salida. Para calcular la derivada de estos pesos se descompone la función de pérdida hasta llegar a ellos. Es decir, es sabido que la función de pérdida,  $\mathcal{L}_W(\hat{y}(x), y)$ , se calcula en base a la salida de la red,  $\hat{y}$ , que se obtiene mediante la ecuación Ecuación 1.6, donde aparecen, a su vez, la función de activación de la capa de salida y la multiplicación que en esta capa se hace de las entradas por los pesos, sobre los que se va a derivar. Este proceso continua hasta llegar a la primera capa. De este modo, se obtiene la derivada de la función de pérdida respecto de todos los pesos de la red.

Por lo tanto, para una red neuronal con  $l$  capas, la derivada del peso  $j$  de la neurona  $i$  de la capa de salida,  $l$ , es

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l}}$$

La derivada anterior no se puede calcular directamente porque la función de pérdida no depende del peso, sino de la salida de la red,  $\hat{y}$ , que es la salida de la última capa,  $\hat{y} = \hat{y}^l$ , sobre la que se seguirá derivando aplicando la regla de la cadena:

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l}} = \frac{\partial \mathcal{L}}{\partial \hat{y}^l} \frac{\partial \hat{y}^l}{\partial w_j^{i,l}}$$

Siendo precisos, el peso está relacionado con la salida  $i$ -ésima de la red (la neurona  $i$  de la capa de salida). Por lo tanto,

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i^l} \frac{\partial \hat{y}_i^l}{\partial w_j^{i,l}}$$

La salida  $i$ -ésima de la red se obtiene a partir de la función de activación de la capa de salida,  $f^l$ , y no del peso, por lo que hay que volver a aplicar la regla de la cadena para derivar la función de activación respecto del peso:

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i^l} \frac{\partial \hat{y}_i^l}{\partial w_j^{i,l}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i^l} \frac{\partial \hat{y}_i^l}{\partial f^l} \frac{\partial f^l}{\partial w_j^{i,l}}$$

Por último, la función de activación se calcula a partir del producto de los pesos de la neurona  $i$  por la entrada de su capa,  $x^l$ ,  $g^{i,l}(x^l) = w^{i,l^T} x^l$ . Esto requiere que se vuelva a aplicar la regla de la cadena por última vez para poder derivar respecto a este producto el peso:

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i^l} \frac{\partial \hat{y}_i^l}{\partial w_j^{i,l}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i^l} \frac{\partial \hat{y}_i^l}{\partial f^l} \frac{\partial f^l}{\partial w_j^{i,l}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i^l} \frac{\partial \hat{y}_i^l}{\partial g^{i,l}} \frac{\partial g^{i,l}}{\partial w_j^{i,l}}$$

Con la expresión anterior ya es posible calcular la derivada  $\frac{\partial \mathcal{L}}{\partial w_j^{i,l}}$ , puesto que se conoce el valor de los términos  $\frac{\partial \mathcal{L}}{\partial \hat{y}_i^l}$ ,  $\frac{\partial \hat{y}_i^l}{\partial f^l}$ ,  $\frac{\partial f^l}{\partial g^{i,l}}$  y  $\frac{\partial g^{i,l}}{\partial w_j^{i,l}}$  donde

$$\frac{\partial g^{i,l}}{\partial w_j^{i,l}} = x_j^l$$

ya que el peso  $j$  de la neurona  $i$  de la capa  $l$  respecto del que se deriva,  $w_j^{i,l}$ , está conectado con la entrada  $j$  de la última capa,  $x_j^l$ , que fue obtenido previamente en el *forward pass*.

Antes de pasar a calcular los pesos de la capa anterior, se observa que, en la expresión que se ha obtenido para calcular la derivada de los pesos de la última capa, existen términos que son comunes para los pesos de todas las neuronas (todos los que no dependen del índice  $j$ ) y que, por tanto, no es necesario que sean calculados más de una vez. Para evitar calcular estos términos de forma redundante en cada peso, se define el concepto de señal de error de una neurona, que recoge los cálculos de la derivada que son independientes del peso sobre el que se deriva. Esta señal de error, que se denota como  $\delta_i^l$  para indicar la señal de error relativa a la neurona  $i$  de la capa  $l$ , se propaga hacia atrás (primero a los pesos de la neurona y después a los pesos de las capas anteriores), de ahí el nombre del algoritmo *backpropagation*. Para la capa de salida, la señal de error es

$$\delta_i^l = \frac{\partial \mathcal{L}}{\partial \hat{y}_i^l} \frac{\partial \hat{y}_i^l}{\partial f^l} \frac{\partial f^l}{\partial g^{i,l}}$$

Ahora se puede evitar los cálculos repetidos y definir la derivada de todos los pesos de la última capa como

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l}} = \delta_i^l \frac{\partial g^{i,l}}{\partial w_j^{i,l}}$$

donde  $\delta_i^l$  ha sido calculado previamente.

Para las capas anteriores a la de salida, el proceso es idéntico. Se calcula la derivada de los pesos respecto a la función de pérdida, pero, al igual que antes, se aprovechan

los cálculos ya realizados retropropagando la señal de error de una capa a la anterior. Esta retropropagación se suele interpretar como que cada neurona es responsable del error que cometan las neuronas a las que envía su salida, por lo que, estas, les devuelven, mediante la señal de error, cómo modificar sus pesos para corregir la salida.

Hay que tener presente que, normalmente, se trabajan con capas *fully connected* donde todas las neuronas de una capa están conectadas con las de la capa anterior. Esto implica que todas las neuronas de la capa de salida han recibido como entrada todas las salidas de las neuronas de la capa anterior. Por lo tanto, al calcular la derivada de un peso de la capa inmediatamente anterior a la de salida, la capa  $l - 1$ , cada peso ha contribuido a la salida de su neurona y esta, a su vez, ha contribuido a todas y cada una de las salidas de la red y deberá calcular la derivada respecto a cada una. Es decir, para un peso  $j$  de la neurona  $i$  de la capa  $l - 1$  su derivada es

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l-1}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_1^l} \frac{\partial \hat{y}_1^l}{\partial w_j^{i,l-1}} + \cdots + \frac{\partial \mathcal{L}}{\partial \hat{y}_{n_l}^l} \frac{\partial \hat{y}_{n_l}^l}{\partial w_j^{i,l-1}}$$

donde  $n_l$  es el número de neuronas de la capa de salida.

Al igual que antes, cada uno de estos términos se desarrollan mediante la regla de la cadena hasta encontrar la expresión que contiene al peso sobre el que se está derivando. Utilizando en la expresión anterior la señal de error se obtiene:

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l-1}} = \delta_1^l \frac{\partial g^{1,l}}{\partial w_j^{i,l-1}} + \cdots + \delta_{n_l}^l \frac{\partial g^{n_l,l}}{\partial w_j^{i,l-1}}$$

En la expresión anterior,  $g^{k,l}$ , hace referencia a la multiplicación de los pesos de la neurona  $k$  de la capa  $l$  por la entrada de esta,  $x^l$ . El peso está en una capa anterior, la capa  $l - 1$ , por lo que hay que seguir expandiendo esta expresión hasta llegar a él. Dado que  $x^l$  no es más que la salida de la capa anterior,  $y^{l-1}$  (junto con el sesgo) y, concretamente, la salida de la neurona del peso que se está derivando es la componente  $i$ , tanto de  $x^l$  como de  $y^{l-1}$ , se puede seguir aplicando la regla de la cadena y derivar sobre la salida de la capa:

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l-1}} = \delta_1^l \frac{\partial g^{1,l}}{\partial x_i^l} \frac{\partial x_i^l}{\partial y_i^{l-1}} \frac{\partial y_i^{l-1}}{\partial w_j^{1,l-1}} + \cdots + \delta_{n_l}^l \frac{\partial g^{n_l,l}}{\partial x_i^l} \frac{\partial x_i^l}{\partial y_i^{l-1}} \frac{\partial y_i^{l-1}}{\partial w_j^{1,l-1}}$$

Derivar a partir de la salida de la red es equivalente a como se hace en las neuronas de la capa de salida solo que con los índices relativos a las capas actualizados:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_j^{i,l-1}} &= \delta_1^l \frac{\partial g^{1,l}}{\partial x_i^l} \frac{\partial x_i^l}{\partial y_i^{l-1}} \frac{\partial y_i^{l-1}}{\partial f^{l-1}} \frac{\partial f^{l-1}}{\partial g^{i,l-1}} \frac{\partial g^{i,l-1}}{\partial w_j^{i,l-1}} + \\ &+ \delta_2^l \frac{\partial g^{2,l}}{\partial x_i^l} \frac{\partial x_i^l}{\partial y_i^{l-1}} \frac{\partial y_i^{l-1}}{\partial f^{l-1}} \frac{\partial f^{l-1}}{\partial g^{i,l-1}} \frac{\partial g^{i,l-1}}{\partial w_j^{i,l-1}} + \\ &+ \cdots + \\ &+ \delta_{n_l}^l \frac{\partial g^{n_l,l}}{\partial x_i^l} \frac{\partial x_i^l}{\partial y_i^{l-1}} \frac{\partial y_i^{l-1}}{\partial f^{l-1}} \frac{\partial f^{l-1}}{\partial g^{i,l-1}} \frac{\partial g^{i,l-1}}{\partial w_j^{i,l-1}} \end{aligned}$$

La señal de error de las neuronas de la capa  $l - 1$  son todos los términos de la derivada que no tienen que ver con el índice del peso y, debido a ello, van a ser el mismo para todos. Para cada neurona  $i$  de la capa  $l - 1$  la señal de error es

$$\delta_i^{l-1} = \left( \sum_{k=1}^{n_l} \delta_k^l \frac{\partial g^{k,l}}{\partial x_i^l} \right) \frac{\partial x_i^l}{\partial y_i^{l-1}} \frac{\partial y_i^{l-1}}{\partial f^{l-1}} \frac{\partial f^{l-1}}{\partial g^{i,l-1}}$$

Utilizando la señal de error, la derivada quedaría como

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,l-1}} = \delta_i^{l-1} \frac{\partial g^{i,l-1}}{\partial w_j^{i,l-1}}$$

que, al igual que en la capa de salida,  $g^{i,l-1}$  es la multiplicación de los pesos de la neurona por la entrada de la capa. Al derivar respecto del peso se tiene que

$$\frac{\partial g^{i,l-1}}{\partial w_j^{i,l-1}} = x_j^{l-1}$$

El procedimiento para calcular la derivada es el mismo para todos los pesos del resto de capas de la red y se puede generalizar con la siguiente expresión:

$$\frac{\partial \mathcal{L}}{\partial w_j^{i,r}} = \delta_i^r \frac{\partial g^{i,r}}{\partial w_j^{i,r}}$$

donde  $\delta_i^r$  es la señal de error relativa a la neurona  $i$  de la capa  $r$  y  $j$  el peso sobre el que se calcula la derivada.

Para los pesos que actualizan el valor del *bias* lo único que cambia es que el valor que multiplican, en lugar de ser una entrada,  $x$ , que viene de la salida de una capa anterior, es directamente el valor del *bias*, 1 normalmente, que se puede reemplazar en la expresión de la derivada. Por ejemplo, para el peso  $j$  de la neurona  $i$  de la capa  $r$ , que modifica el valor del *bias*, al derivarlo respecto de  $g^{i,r}$  se tiene que

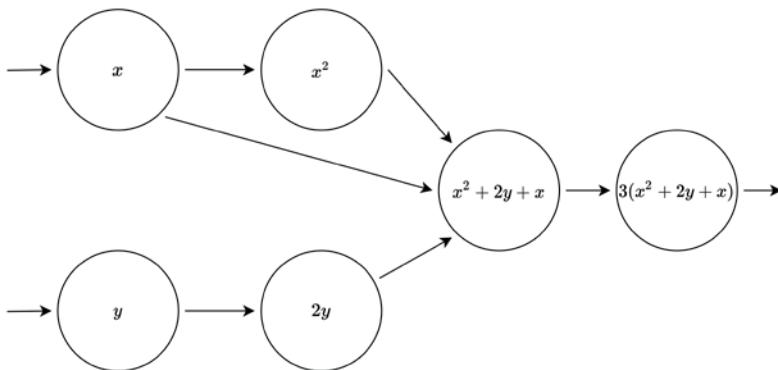
$$\frac{\partial g^{i,r}}{\partial w_j^{i,r}} = x_j^r = 1$$

La complejidad de este algoritmo radica en la multitud de índices implicados en los cálculos, que dificultan su comprensión. No hay que olvidar que lo único que hace *backpropagation* es calcular la derivada de la función de pérdida respecto de cada peso y que, para ahorrar cálculos redundantes, se utiliza el concepto de señal de error para propagar los que se puedan reutilizar.

En principio, este algoritmo no está limitado a trabajar con ningún tipo de función de activación ni operación concreta. Se pueden utilizar cualquier combinación de operaciones en la red, siempre y cuando sean derivables, ya que el ajuste de los pesos requiere de ir derivando sobre las operaciones que generan la salida de la red hasta llegar al peso que se está actualizando.

Es común representar los cálculos de *backpropagation* sobre un grafo de operaciones o *computational graph* en inglés. Esto no es más que representar una serie de operaciones como un grafo acíclico dirigido donde cada nodo representa una

operación que se realiza sobre la salida de los nodos que se conectan a este. En la Figura 2.24 se muestra un ejemplo del grafo de operaciones para la operación  $f(x, y) = 3(x^2 + 2y + x)$ .



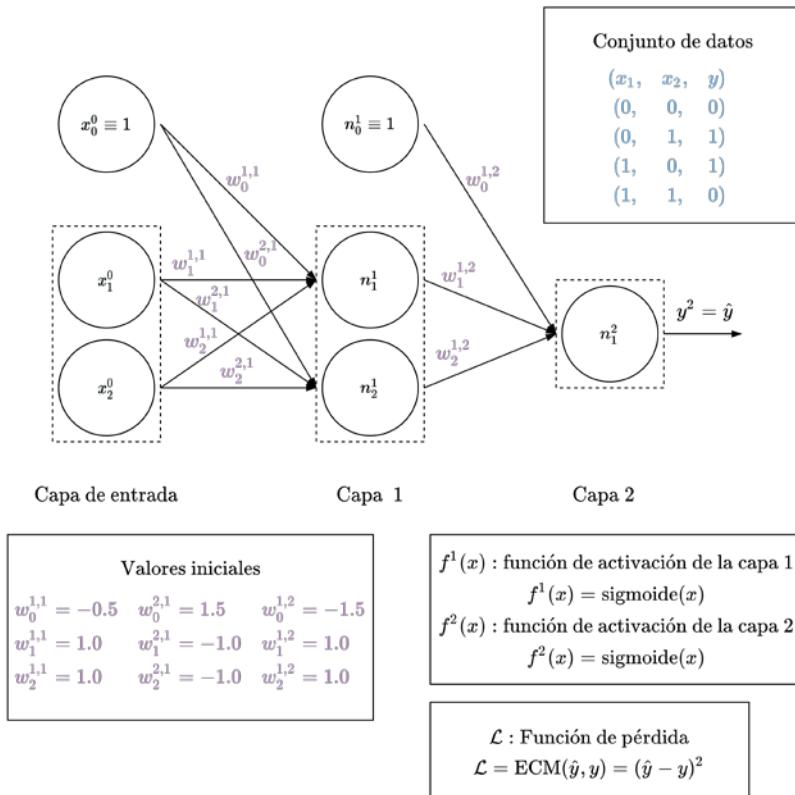
**Figura 2.24:** Ejemplo de grafo de operaciones

Este grafo muestra de forma simbólica las operaciones que se van a realizar sobre las entradas,  $x$  e  $y$ . Con un grafo computacional se puede representar también todas las operaciones implicadas en el cálculo del valor de pérdida, lo que incluye, evidentemente, a la propia función de pérdida y la red neuronal como nodos del grafo.

En un principio, se realiza el *forward pass* con los datos de ejemplo y se van obteniendo las salidas de cada nodo. Cuando se llega a obtener el valor de pérdida para los datos de entrada actuales, comienza el *backward pass* donde se van calculando las derivadas en cada nodo. Además de calcular las derivadas, se calculan y envían los términos comunes como señal de error a los nodos anteriores al actual. Cuando se llega a los nodos en los que se utilizan los pesos, se calculan sus derivadas a partir de la señal de error recibida y cada uno actualiza su valor en base a la regla de actualización del algoritmo de optimización elegido y el valor de su derivada.

Para finalizar esta sección y a modo de resumen, en la Figura 2.25 se representa una red para el problema de ajustar la función XOR mediante dos capas con dos y una

neurona respectivamente. Las funciones de activación para cada capa, el conjunto de datos, los valores iniciales y la función de pérdida a utilizar se detallan en la propia figura.



**Figura 2.25:** Ejemplo completo de red neuronal para el problema XOR

A partir de la Figura 2.25 se puede generar el grafo de operaciones equivalentes sobre el que se aplica el algoritmo *backpropagation* para calcular las derivadas de los pesos y actualizarlos con el algoritmo descenso por gradiente utilizando la función de pérdida. Este grafo se representa en la Figura 2.26.

Sobre el grafo anterior, en la Figura 2.27 se calcula el *forward pass* (color verde) y el *backward pass* (color rojo) de una iteración del algoritmo.

Como se observa en la Figura 2.27, el error que comete la red para el ejemplo  $(1, 1, 0)$  con el valor inicial de los pesos (ver Figura 2.25) es 0.18.

Si se utiliza la derivada que se ha calculado de cada peso para actualizarlos según la regla de actualización (el *learning rate* se establece a 1)

$$W_{t+1} = W_t - 1 \cdot \nabla_W \mathcal{L}_{W_t}(\hat{y}, y)$$

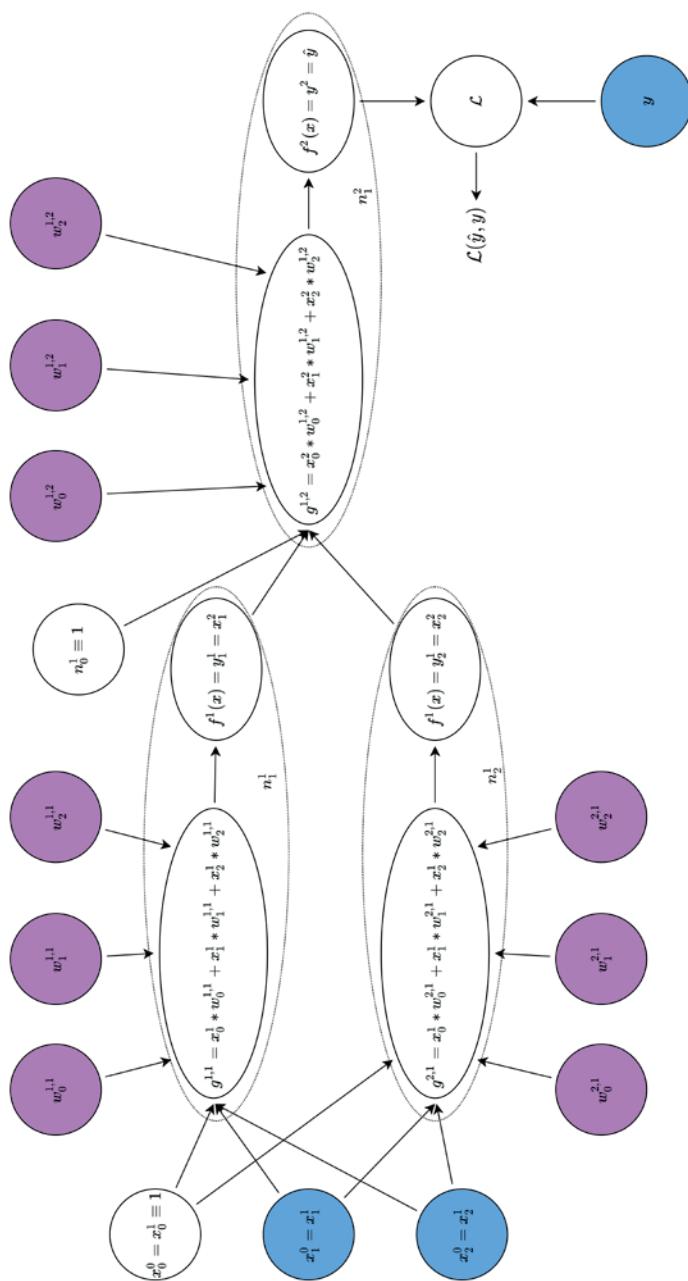
se tiene que los nuevos valores de los pesos son

$$\begin{aligned} w_0^{1,1} &= -0.5 - 0.03 = -0.53 & w_0^{2,1} &= 1.5 - 0.048 = 1.452 \\ w_1^{1,1} &= 1.0 - 0.03 = 0.97 & w_1^{2,1} &= -1.0 - 0.048 = -1.048 \\ w_2^{1,1} &= 1.0 - 0.03 = 0.97 & w_2^{2,1} &= -1.0 - 0.048 = -1.048 \\ w_0^{1,2} &= -1.5 - 0.207 = -1.707 \\ w_1^{1,2} &= 1.0 - 0.17 = 0.83 \\ w_2^{1,2} &= 1.0 - 0.078 = 0.922 \end{aligned}$$

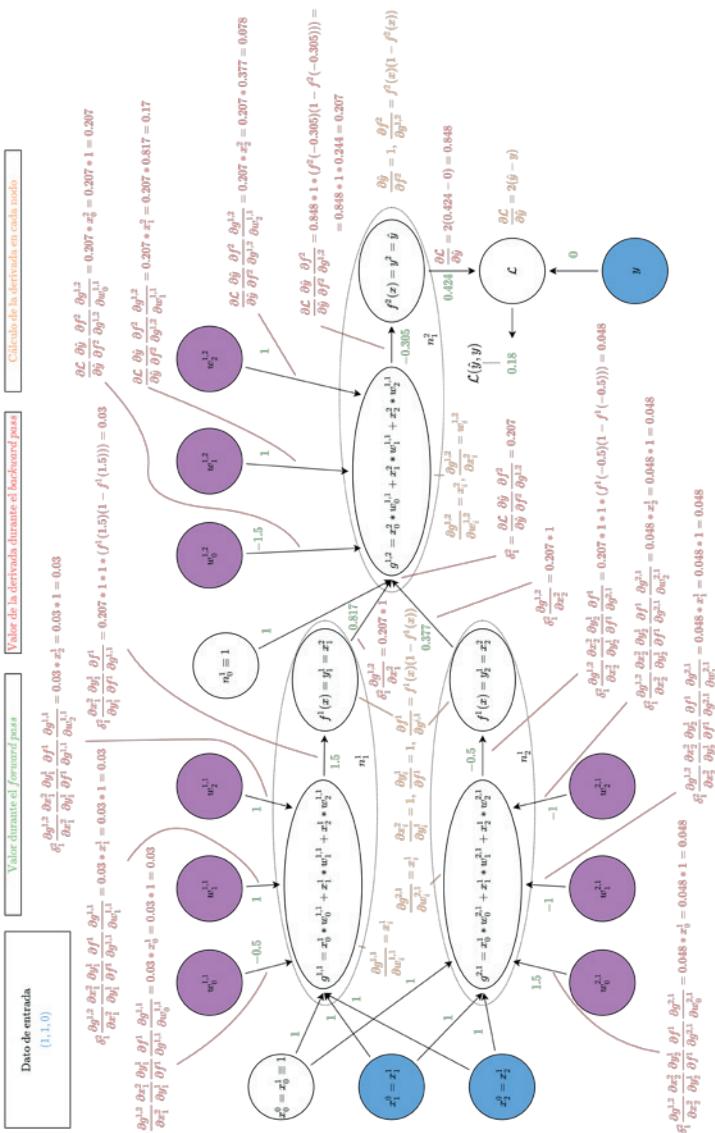
Si se utilizan estos pesos para volver a calcular la pérdida para el mismo dato de entrada,  $(x_1, x_2, y) = (1, 1, 0)$ , se obtiene un valor de pérdida de 0.107. Por lo tanto, utilizando el algoritmo *backpropagation* para calcular la derivada de los pesos, y actualizándolos en base a la regla del descenso por gradiente, se ha conseguido pasar de un valor de pérdida de 0.18 a 0.107, se ha reducido el valor de pérdida en 0.073.

En la práctica, la mayoría de las librerías utilizan esta forma de representar el problema de optimización para calcular de manera simbólica las derivadas que después se utilizan en el cálculo del gradiente cuando se entrena la red con un conjunto de datos. Como ya se ha indicado, *backpropagation* podría trabajar con cualquier

grafo de operaciones, representen una red neuronal o no, y con cualquier tipo de operación en los nodos, siempre y cuando estas sean derivables.



**Figura 2.26:** Grafo de operaciones para la red neuronal de la Figura 2.25. En color azul los datos de entrada y salida, y, en morado, los pesos de la red.



**Figura 2.27:** Cálculo de una iteración del algoritmo descenso por gradiente sobre el grafo de operaciones de una red.

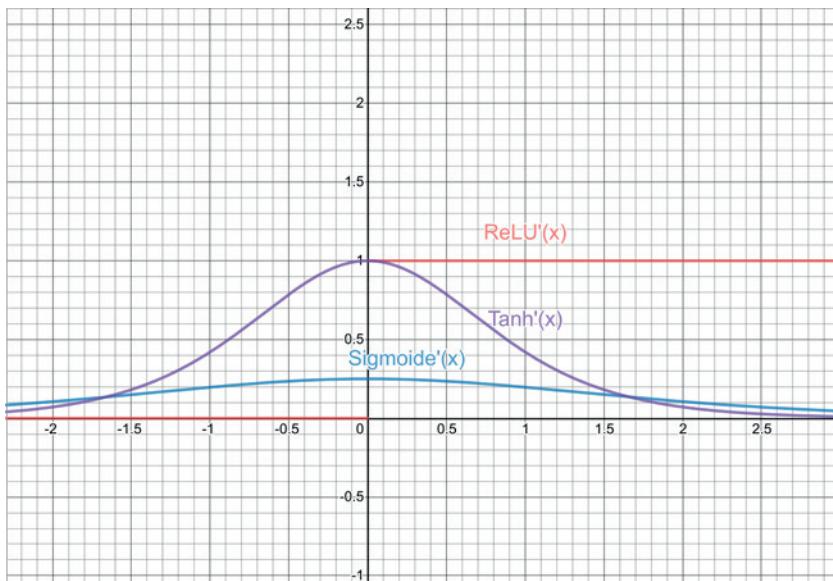
## 2.6.4. Problemas del aprendizaje basado en gradiente

A pesar de que los algoritmos basados en el uso del gradiente para actualizar los pesos de una red son ampliamente utilizados hoy día debido a sus resultados, hay que tener presente que esta forma de actualizar los pesos presenta algunos inconvenientes que se van a estudiar a continuación.

El primer problema que se encuentra al utilizar la derivada para actualizar los pesos tiene que ver con la forma en la que se calcula. Al aplicar la regla de la cadena, lo que se hace es añadir nuevos términos a la expresión del cálculo de la derivada que multiplican a los anteriores. Cuando se calcula la derivada de un peso de las primeras capas de una red se puede llegar a tener que realizar multitud de multiplicaciones que pueden producir la **explosión del gradiente** (*exploding gradient* en inglés) o su **desvanecimiento** (*vanish gradient* en inglés). La explosión del gradiente ocurre cuando, en esta cadena de multiplicaciones, los términos implicados son mayores que la unidad. Cuando esto ocurre, el valor de la derivada irá creciendo con cada sucesiva multiplicación hasta llegar a valores tan grande que pueden acabar siendo infinitos, por lo que carece de sentido utilizarlos para actualizar los pesos. Por el otro lado, cuando los términos de la multiplicación son menores a la unidad, la secuencia de multiplicaciones irá reduciendo el valor de la derivada con cada multiplicación hasta llegar a valer cero o un valor cercano, lo que hace imposible mover el valor del peso al ser su derivada prácticamente nula.

Uno de los términos implicados en el cálculo de las derivadas de los pesos es la derivada de la función de activación. Según la función de activación que se elija, se puede estar facilitando o dificultando el entrenamiento de la red. En la Figura 2.28 se representan de forma gráfica las derivadas de las principales funciones de activación.

Como se observa en la Figura 2.28, si se utiliza la función de activación sigmoide, solo se consigue valores diferentes de cero cerca del origen, aunque son valores alejados de la unidad. Esto va a provocar que, por cada vez que se derive por esta función, se reduzca el valor de la derivada que se está calculando. La función tanh alcanza valores más alto en el origen pudiendo llegar a la unidad, pero fuera del origen los valores son cercanos a cero. Al contrario que las dos anteriores, la función



**Figura 2.28:** Gráfica de las derivadas de las principales funciones de activación

ReLU no modifica el gradiente cuando la derivada es sobre un valor positivo (al multiplicar por la unidad); como contrapartida, para los valores negativos, el valor de la derivada es cero y no existe información para modificar el peso.

En el caso de que el gradiente se mantenga en un rango de valores aceptables, se podrá ir ajustando los pesos de la red sin sufrir de problemas por su desvanecimiento o su explosión. Sin embargo, independientemente de que se padezcan estos problemas, en alguna iteración del entrenamiento se podría alcanzar un mínimo local y, por lo tanto, el gradiente valdrá cero. No obstante, es difícil que se de esta situación. Primero, hay que tener presente que la superficie de la función de pérdida cambia con cada lote de ejemplos que se utilizan, por lo que, es de esperar que, al cabo de un cierto número de iteraciones, el mínimo que marcan los pesos actuales en la superficie deje de serlo y puedan moverse. Aun así, puede que el *learning rate* que se utilice haga que los pesos apenas se muevan de donde estaban y, al llegar un nuevo lote de ejemplos, se vuelva a quedar parado en el mínimo anterior, produciéndose una especie de oscilación alrededor de este mínimo sin llegar nunca

a moverse fuera de él. A pesar de ello, hay que tener presente que, normalmente, una red neuronal puede tener miles sino millones de parámetros, que definen la superficie de la función de pérdida. Esto significa que el mínimo que se encuentre, aunque puede no ser un mínimo global y sí un punto de silla, muchas de las millones de derivadas han apuntado hacia él. Han coincidido en definir el punto en el que se encuentra los pesos de la red como un mínimo. Esto hace pensar, y así lo apuntan algunos estudios como [13] y [14], que, cuando se encuentra un mínimo local en un espacio de tantas dimensiones, lo más probable es que el mínimo local sea parecido al mínimo global y, por lo tanto, el hecho de caer en un mínimo local no es un problema que deba preocupar.

Otro de los problemas relacionados con el uso del gradiente es el conocido como **olvido catastrófico** (*catastrophic forgetting* en inglés). Este problema tiene que ver con la dificultad que tienen las redes de mantener el desempeño conseguido en un problema al ser entrenado en uno nuevo. Los humanos, por ejemplo, pueden aprender a montar en bici y después a conducir un coche sin olvidar cómo se monta en bici. Las redes neuronales, por el contrario, cuando son entrenadas en una nueva tarea, olvidan cómo resolver la primera en la que fueron entrenadas. Esto se debe a la forma en la que se actualizan los pesos. Esta actualización viene marcada por el gradiente de la superficie de la función de pérdida y esta superficie la definen los datos de ejemplo de la nueva tarea y no los de la anterior. Por lo tanto, el valor de pérdida a partir del que se deriva no penaliza el error que comete la red en la primera tarea al no tener datos de esta. La red intenta aproximar los nuevos datos de ejemplo con todas las consecuencias, incluida la pérdida de la información previa en los pesos que le permitía resolver la primera tarea en la que fue entrenada.

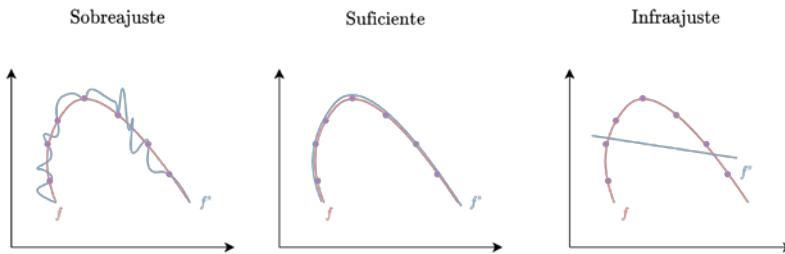
Como se vio en la Sección 1.3.3, las redes neuronales son tratadas como cajas negras de las que no se sabe el porqué de su funcionamiento, puesto que este se definió únicamente con el objetivo de minimizar el error que comete la red ante unos datos de ejemplo. Sin embargo, la solución encontrada para minimizar el error en los datos del conjunto de entrenamiento no tiene porqué funcionar para nuevos datos. Como ya se vio en la citada sección, las redes neuronales pueden fallar ante entradas nunca vistas, incluso cuando se realizan leves modificaciones a uno de los datos del conjunto de entrenamiento. Esto es un caso de **sobreajuste** (*overfitting* en inglés) de la red y es un problema inherente a cualquier modelo de *machine learning*. Se produce sobreajuste cuando el modelo no es capaz de extrapolar sus

resultados a datos diferentes a los del conjunto de entrenamiento. El caso contrario al sobreajuste es el **infraajuste** (*underfitting* en inglés), que se da cuando el modelo no es capaz ni de dar buenos resultados en el conjunto de datos de entrenamiento. Para entender la gravedad de este problema en las redes neuronales, estudios como [15] demuestran que una red neuronal es capaz de clasificar correctamente un conjunto de imágenes cuyas clases han sido generadas de manera aleatoria. Esto da una idea de lo presente que hay que tener el problema del sobreajuste cuando se entrena las redes, ya que se corre el riesgo de que se aprendan los datos del problema de memoria y fallen estrepitosamente ante datos nunca vistos.

Los problemas de sobreajuste e infraajuste suelen estar relacionados con la capacidad expresiva del modelo. A mayor número de neuronas y capas, mayor complejidad podrá tener la función que puede aproximar la red. Por el contrario, si hay un número inferior de neuronas o de capas al necesario para aproximar la función descrita por los datos del conjunto de entrenamiento, la red no logrará buenos resultados. El ejemplo más evidente del último caso es cuando se intenta aproximar la función XOR con un único perceptrón. El modelo sufrirá infraajuste, ya que no tiene capacidad suficiente para aproximar la función objetivo. Cuando el modelo sufre sobreajuste significa que, debido a que tiene capacidad más que suficiente para aproximar la función objetivo, le es más fácil de resolver el problema ajustando los pesos para que approximen los datos del conjunto de entrenamiento que intentar buscar la posible distribución que subyacen en los datos y que es la que se quiere aproximar. Esto provoca que la red construya una función bastante más compleja que la que se intenta aproximar para ajustarse únicamente a los datos que ha recibido.

Para comprender mejor los problemas de sobreajuste e infraajuste, en la Figura 2.29 se muestra un ejemplo de tres modelos donde se observa la función a aproximar (color rojo),  $f(x)$ , y la función aproximada por cada modelo (color azul),  $f^*(x)$ , en base a los puntos de ejemplo que se les ha suministrado durante el entrenamiento (color morado). Se observa que el modelo central es el que se ajusta mejor a la curva que parece que describen los datos. El primer modelo, aunque se ajusta perfectamente a los datos de ejemplo, lo hace describiendo una curva bastante compleja que dista mucho de la curva que describen los datos. El último modelo no tiene capacidad suficiente para aproximar los puntos de ejemplo.

Cuando el modelo logra tener un buen comportamiento (modelo central de la



**Figura 2.29:** Ejemplo de sobreajuste (izquierda), buen ajuste (centro) e infraajuste (derecha) en base a un conjunto de datos de ejemplo (color morado). En color rojo se indica la función objetivo y en color azul la función descrita por cada modelo.

Figura 2.29) ante datos nunca visto se dice que ha sido capaz de generalizar a partir de los datos de ejemplo. Es decir, a partir de unos datos de muestra, ha logrado obtener un modelo que le permite resolver el problema para datos diferentes a los del entrenamiento. El hecho de que se consiga que una red generalice tiene que ver tanto con la capacidad de esta como con lo representativo que sea el conjunto de datos con el que se entrena.

La falta de generalización de un modelo es debida a que comete cierto error al aproximar una función. Es decir, al intentar aproximar la función,  $f$ , con una red neuronal,  $f^*$ , no siempre se logra aproximar exactamente la función objetivo, sino que se comete cierto error,  $\epsilon$ , de forma que  $f = f^* + \epsilon$ . Si se asume que parte del error es debido a un cierto error irreducible por problemas inherentes al propio conjunto de datos (*outliers* o ruido que haga a estos no representar correctamente un punto de la función original), el resto del error que comete el modelo suele presentar una distribución normal con cierta media y desviación típica. El error para una entrada,  $x$ , se define como el valor esperado de la diferencia al cuadrado entre la salida de la red y la salida esperada,  $\epsilon(x) = E[(f^*(x) - y)^2]$ , y, a partir de este, se calcula el sesgo como  $E(f^*(x)) - y$ ; y la varianza como  $E[(f^*(x) - E[f^*(x)])^2]$ . Al descomponer la función de error se llega a que este puede explicarse en base a estos dos estimadores como  $\epsilon(x) = \text{varianza}(x) + \text{sesgo}^2(x)$ . Cuando el error que comete el modelo es debido a una falta de capacidad (infraajuste) se habla de que el modelo tiene un sesgo alto, ya que, sin importar el dato de entrada, el mode-

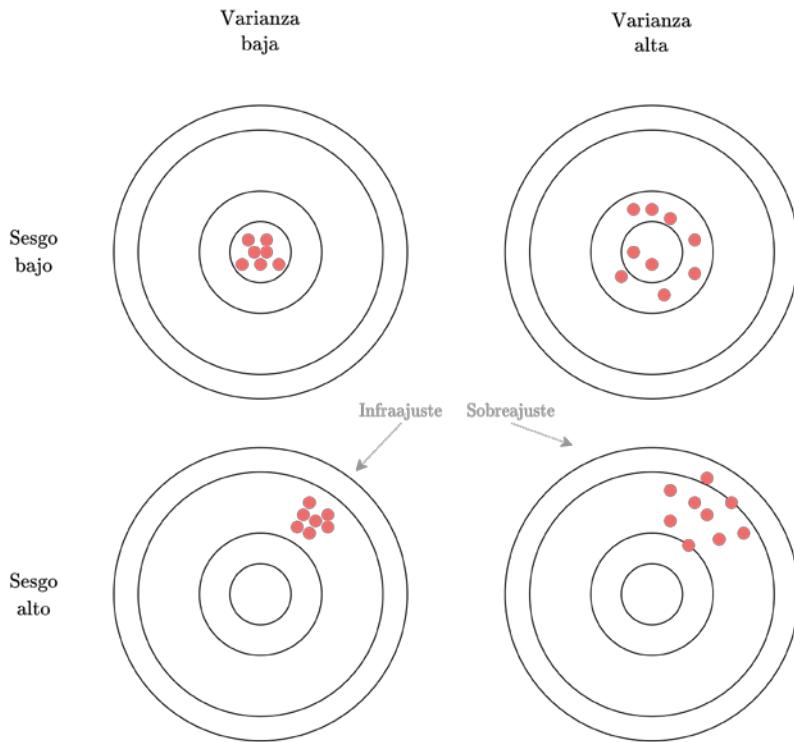
lo va a tener un error de base que no puede evitar. Si el problema es que el modelo es muy complejo, con demasiada capacidad, el error irá variando con cada dato de entrada, puesto que algunos serán parecidos a los datos que ha sobreajustado el modelo y el error será bajo. Aquellos que disten de los aprendidos por el modelo producirán un error alto. En este caso, se dice que el modelo tiene una varianza alta.

Para comprender mejor los conceptos de sesgo y varianza, en la Figura 2.30, se muestra un problema de ejemplo en el que se representan las predicciones del centro de una diana hechas por modelos que sufren todas las posibles combinaciones de estos dos tipos de errores. Cuando el sesgo es alto, no se logra acertar al centro de la diana, ya que siempre hay un error de base. Por otro lado, cuando la varianza es alta, aunque algunos puntos acierten, muchos de ellos se alejan del centro. Lo ideal es que ambos errores sean bajos y parejos.

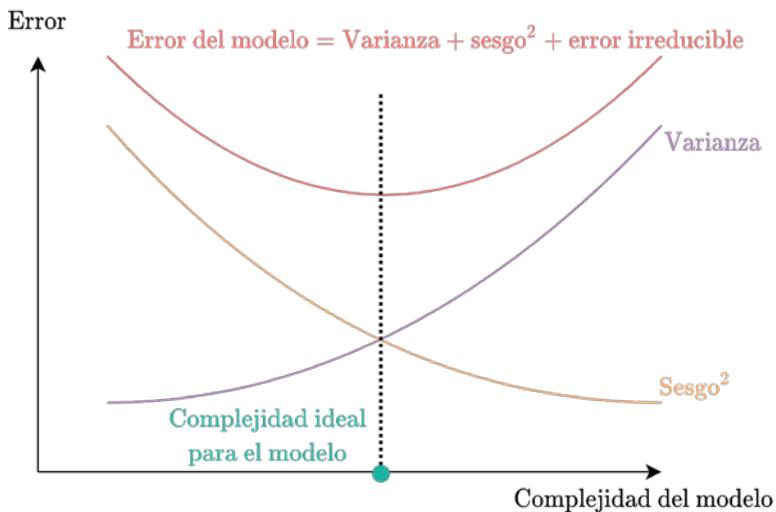
Normalmente, estos dos errores son incompatibles. Cuando se intenta reducir el sesgo incrementando la complejidad del modelo se puede aumentar la varianza y viceversa. Este dilema es lo que se conoce como el compromiso entre sesgo y varianza, que se intenta alcanzar a la hora de diseñar la arquitectura de la red para un problema. En la Figura 2.31 se muestra de forma gráfica el efecto que tiene la complejidad de la red en el error que comete por sesgo y varianza así como el punto de equilibrio ideal. Este es el punto en el que se debe buscar situar la complejidad del modelo cuando se define su arquitectura para reducir todo lo posible ambos errores sin que se aparezca un problema en el comportamiento del modelo.

Como se observa en la Figura 2.31, la definición de la complejidad del modelo, es decir, la arquitectura de la red, tiene que ser la adecuada para el problema que se está abordando si no se quiere padecer de un sesgo o varianza altos.

En las siguientes secciones se intentan dar algunas pautas para lograr un entrenamiento exitoso y evitar los problemas mencionados en esta sección.



**Figura 2.30:** Posibles escenarios de un modelo respecto al sesgo y la varianza



**Figura 2.31:** Búsqueda de compromiso entre sesgo y varianza al definir la complejidad del modelo

## 2.7. Parámetros del entrenamiento

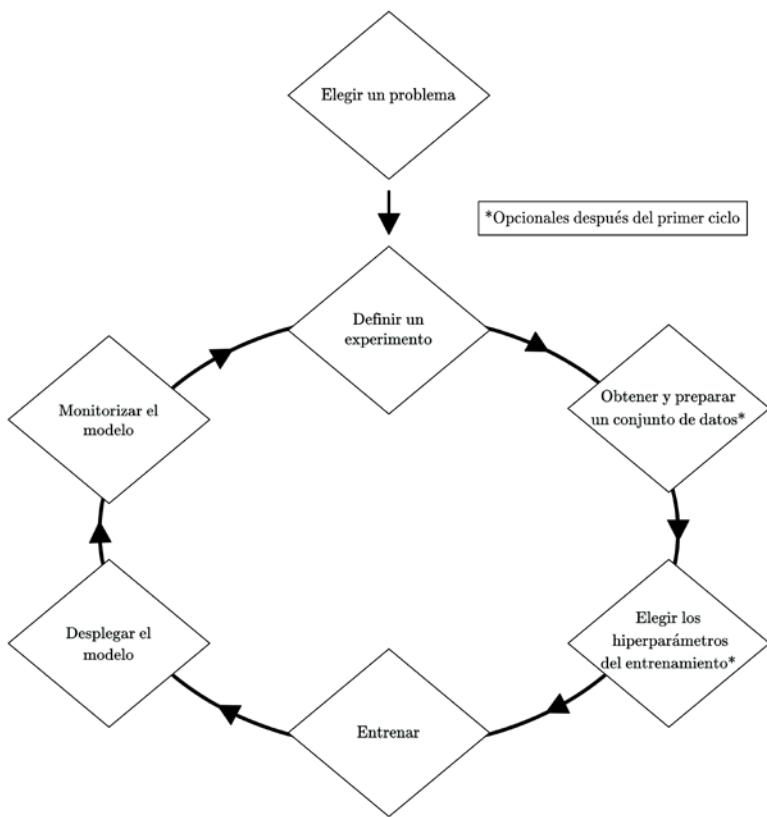
El entrenamiento de una red neuronal conlleva la elección de multitud de parámetros incluso antes de que comience el entrenamiento. Desde la arquitectura de la red hasta el *learning rate* o el número de ejemplos que se van a utilizar en el lote de entrenamiento son algunos ejemplos de los parámetros a elegir de antemano, sin una idea clara del efecto en los resultados del entrenamiento para los valores escogidos. A estos parámetros se les llaman **hiperparámetros**, puesto que son los parámetros que definen a su vez el entrenamiento que permitirá encontrar los valores para los parámetros (pesos) de la red.

Los hiperparámetros suelen elegirse en base a diferentes criterios como la experiencia previa, el estudio del estado del arte para problemas similares al que se tratar de resolver o pequeñas pruebas que se realicen sobre el conjunto de datos de entrenamiento. Aquellos que mejores resultados muestren serán con los que se hará un entrenamiento más extenso de la red. No obstante, un único entrenamiento no suele ser suficiente para conseguir los mejores resultados. Se suelen desarrollar diferentes experimentos de forma cíclica con la idea de ir mejorando poco a poco los resultados del experimento anterior, ya sea por probar nuevos hiperparámetros o centrarse en la recolección de más datos y en mejorar la calidad de estos.

El desarrollo de redes neuronales suele tener un ciclo de vida como el representado en la Figura 2.32 donde, una vez definido el problema que se va a tratar, se obtienen y preparan los datos y se definen los hiperparámetros para el entrenamiento que, una vez finalizado, obtiene los pesos con los que hacer el despliegue en producción de la red y se monitoriza su comportamiento. En base a la monitorización, se podrá definir un nuevo experimento para mejorar sus resultados.

Tal y como se observa en la Figura 2.32, el desarrollo de redes neuronales es un proceso iterativo en el que la elección de los hiperparámetros, así como de los datos, juegan un papel esencial en el entrenamiento de la red.

Además de la división en experimentos por cada prueba de hiperparámetros o nuevos datos que se haga, la etapa de entrenamiento se suele dividir en un número de **épocas**. Una época consiste en utilizar todos los datos del conjunto de entrenamiento para entrenar una red. Cada época viene definida por el número de **iteraciones** que necesita el algoritmo de optimización para utilizar todos los datos del conjunto de entrenamiento. A su vez, una iteración se compone de tantos ejemplos

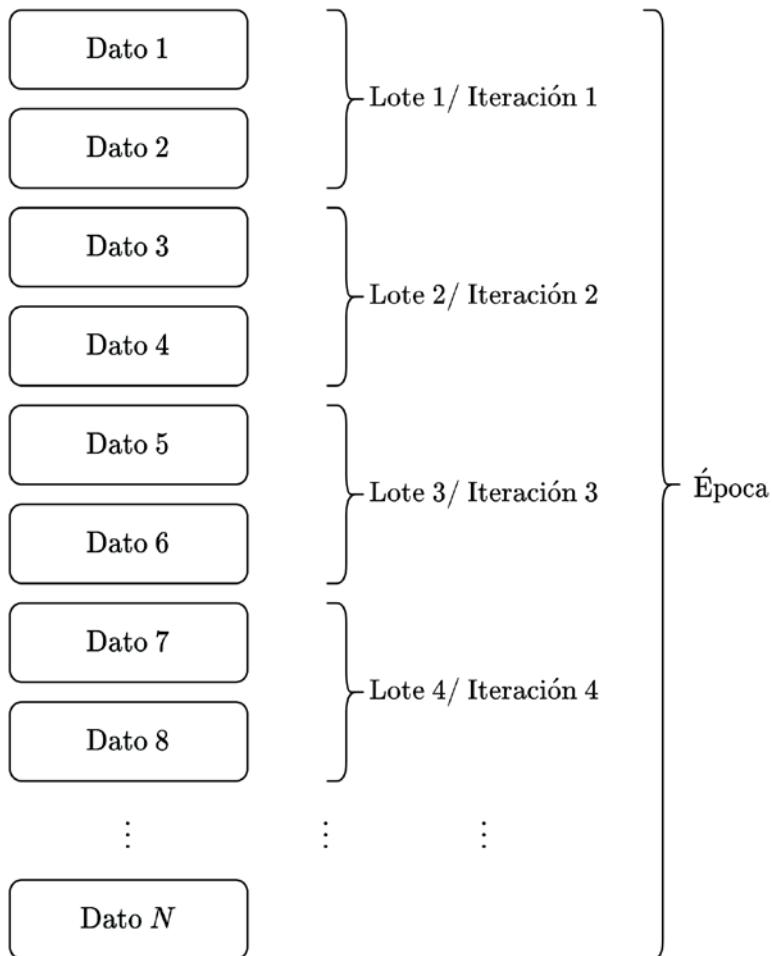


**Figura 2.32:** Ciclo de vida del desarrollo de redes neuronales

como se haya definido el tamaño del **lote** para el algoritmo de entrenamiento. En la Figura 2.33 se muestra la relación entre todos estos conceptos.

Al finalizar cada época es cuando se evalúa la red para obtener las métricas sobre el conjunto de validación. También se suele aprovechar el final de la época para comprobar si alguna de las métricas ha mejorado respecto a los mejores resultados obtenidos hasta el momento durante el entrenamiento. En el caso de que haya alguna mejora se almacena el valor de los pesos actuales como los mejores valores para los pesos de la red en el entrenamiento en base a la métrica que se ha mejorado.

Conjunto de datos



**Figura 2.33:** Relación entre los datos de un conjunto de datos, el tamaño del lote, la iteración y la época de entrenamiento

Si la validación es costosa, también se puede optar por almacenar los pesos después de cada época y, al acabar el entrenamiento, elegir la mejor de las redes en base a los resultados que consigan en el conjunto de validación. Si tampoco es posible almacenar todos los pesos al final de cada época por el tamaño que ocupan, se puede almacenar cada cierto número de épocas en lugar de en todas.

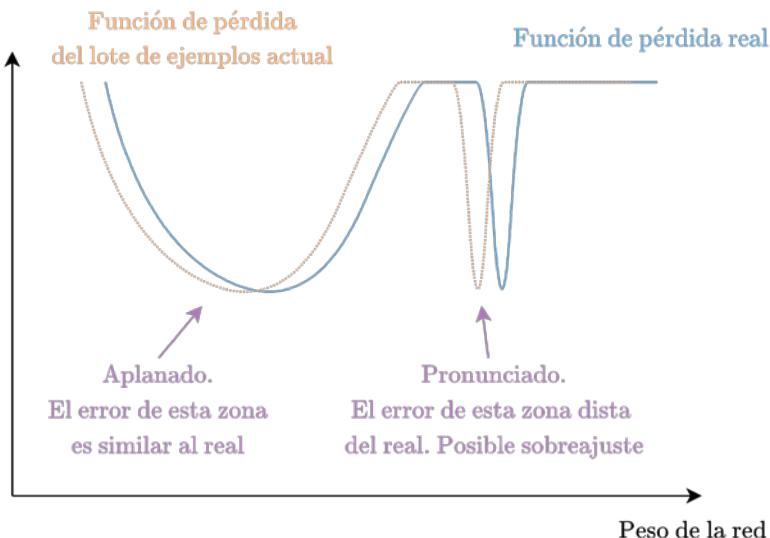
A continuación, se detallan los principales hiperparámetros involucrados en el entrenamiento de una red junto con algunas recomendaciones a la hora de definirlos:

- **Arquitectura de la red.** La elección de la arquitectura de la red definirá su capacidad, que deberá ser acorde al problema que se está resolviendo. No se trata de una elección fácil, ya que hay que definir el número de capas, las neuronas en cada una y la función de activación a utilizar. La opción más sencilla consiste en utilizar la misma arquitectura que la de alguna red que se haya empleado en un problema similar con resultados contrastados. Sin embargo, no hay ninguna garantía de que una red que ha funcionado bien en un problema similar lo vaya a hacer igual de bien. Esto es lo que viene a decir el teorema conocido como *no free lunch* [16]. Según este teorema, no existe un modelo que vaya a conseguir mejores resultados que todos los demás para cualquier problema. Es algo que se puede intuir: si se utiliza la arquitectura de una red diseñada para un problema complejo, al aplicarla a un problema sencillo, puede que sobreajuste y no consiga tan buenos resultados como una arquitectura más simple. Esto debe animar a buscar o, al menos, a modificar la arquitectura base de la red cuando se vaya a aplicarla a un nuevo problema para que se ajuste a la complejidad requerida por este. Por ejemplo, no tiene sentido utilizar una red neuronal con cientos de capas y neuronas para abordar el problema de la función XOR, que se puede resolver con tres neuronas organizadas en dos capas.
- **Técnica de inicialización de los pesos.** Tal y como se estudió en la Sección 2.6.1, existen diferentes alternativas para inicializar los pesos de la red y será un hiperparámetro más a elegir. Siempre que sea posible, se intentará utilizar la técnica de *transfer learning* para inicializar la red con los pesos obtenidos en algún entrenamiento previo y, de esta forma, tener más probabilidades de comenzar en una zona cercana al mínimo con el objetivo de que el entrenamiento sea más rápido que si se hace una inicialización puramente aleatoria.
- **Número de épocas del entrenamiento.** Un entrenamiento se compone de un determinado número de épocas. Si el entrenamiento tiene un número de épocas

muy bajo puede que no se llegue a reducir el valor de pérdida lo suficiente como para empezar a obtener buenos resultados. Dependiendo de la complejidad del problema se necesitarán más o menos épocas de entrenamiento, aunque un par de épocas no suele ser nunca suficiente para que el algoritmo se acerque al mínimo.

- **Tamaño del lote de ejemplos.** El número de ejemplos con el que se calcula el valor de pérdida influirá en la forma de la superficie de la función de pérdida y en la dificultad que tendrá el algoritmo de optimización para alcanzar el mínimo. Se suele emplear potencias de dos para definir el tamaño del lote siendo los valores 16, 32, 64 y 128 los más habituales. La elección de uno u otro dependerá de la potencia del equipo que se utilice para hacer el entrenamiento. El tamaño del lote se ve limitado por la memoria que haya disponible y suele ser el motivo por el que se eligen valores por debajo de 32 o 16.
- **Almacenamiento de los pesos de la red.** El experimento que se realice busca obtener los mejores pesos de la red para el problema y, por lo tanto, hay que decidir cuándo y bajo qué criterios se guardan los valores de los pesos para poderlos utilizar en la etapa de inferencia. Como se comentó anteriormente, lo ideal es almacenar los pesos cada vez que se mejore el valor de pérdida o alguna de las métricas del conjunto de validación.
- **Algoritmo de optimización.** *Backpropagation* es la única opción que hay para calcular la derivada de los pesos en las capas ocultas de la red, pero la forma en la que se utilice la derivada para actualizar los pesos viene indicada por el algoritmo de optimización elegido. En la Sección 2.6.2 se estudiaron los principales algoritmo de optimización y las ventajas de cada uno. Además de atender a las bondades de cada algoritmo, hay que tener presente que algunos algoritmos, como descenso por gradiente, son más rápidos de calcular que Adam, por ejemplo. Por otro lado, una de las ventajas del algoritmo descenso por gradiente, y por la que se suele optar por este algoritmo, es que, al ser más simple que otros algoritmos, si se detiene porque han llegado a un mínimo es más probable que este sea un mínimo aplanado y no uno pronunciado. La ventaja de los mínimos aplanados frente a los pronunciados es que tienen más probabilidades de coincidir con el mínimo real que se está buscando (hay que tener presente que el mínimo se define por el lote de ejemplos actual y no por todo el conjunto de datos del problema real). Esto es debido a que un mínimo aplanado significa que incluso si se modifica levemente el valor de los pesos seguirán situados en

el mínimo, mientras que un mínimo pronunciado deja de serlo al mover el valor de los pesos. En la Figura 2.34 se muestra un ejemplo de estos dos tipos de mínimo.



**Figura 2.34:** Diferencias entre un mínimo pronunciado y aplanado de una función de pérdida definida por un lote de ejemplos y la definida por todos los datos

- **Parámetros del algoritmo de optimización.** El algoritmo de optimización escogido suele requerir a su vez definir algunos parámetros. El *learning rate* es uno de los parámetros más importante a escoger, no ya del propio algoritmo, sino de todo el entrenamiento, ya que de él dependerá, en gran medida, si se logra alcanzar el mínimo o no. La mayoría de los otros parámetros de los algoritmos de optimización se suelen dejar al valor por defecto que proponen los autores, pero la elección del *learning rate* requiere de un cuidado especial. Su valor estará relacionado con el tamaño del lote. A mayor número de ejemplos para un lote más probabilidad de que el gradiente indique el mínimo real de la función y, por tanto, se pueda dar un paso más grande en cada iteración usando un *learning rate*.

te alto. Por el contrario, si se dispone de pocos ejemplos, el gradiente contendrá bastante ruido y no marcará el mínimo real, sino alguna zona cercana, por lo que hay que dar pasos de menor tamaño. No es de fiar la dirección marcada por una sola iteración, sino que serán necesarias más iteraciones para llegar a una posición. Es el precio que hay que pagar al utilizar menos ejemplos, aunque, por otro lado, este ruido que tiene el gradiente puede ayudar a escapar de mínimos locales como ya se ha estudiado. Por norma general, se utiliza un *learning rate* de 0.1 para un tamaño del lote de 128 ejemplos y se divide entre 10 cada vez que se divide por 2 el tamaño del lote. Es decir, para 64 ejemplos se usaría un *learning rate* de 0.01 y de 0.001 para 32 ejemplos. Si el valor del *learning rate* es demasiado pequeño, el algoritmo puede no alcanzar nunca el mínimo y, si es demasiado grande, puede acabar divergiendo al dar pasos demasiado grandes y alejarse del mínimo. Serán necesarias algunas pruebas para garantizar de que el entrenamiento puede llevarse a cabo con el valor escogido.

- **Métricas.** Existen multitud de métricas de evaluación, algunas de ellas específicas para determinados problemas. En base al objetivo que se persiga, se deben elegir las métricas con las que se va a evaluar la red, ya que serán las que indiquen cuándo guardar los pesos que luego son utilizados en la etapa de inferencia. Por ejemplo, si se está desarrollando un sistema para el diagnóstico de enfermedades, quizás interesen métricas que tengan más en cuenta si el sistema da falsos negativos (pacientes con la enfermedad clasificados como sanos) que falsos positivos (pacientes sin la enfermedad clasificados como enfermos), puesto que el médico puede determinar después si el paciente está realmente enfermo o no, pero no lo hará si ha sido descartado de antemano al clasificarlo como sano estando enfermo.
- **Función de pérdida.** Aunque en la sección 2.5 se habló sobre las dos funciones de pérdidas más utilizadas (el error cuadrático medio y el *cross-entropy*), estas no son las únicas opciones que se disponen. Como se sabe, el ECM es el más indicado para problemas de regresión y CE para clasificación. No obstante, se puede modificar estas funciones para adaptarlas a la casuística del problema que se esté abordando. Por ejemplo, si el problema tiene muchos *outliers* en el conjunto de datos, la función ECM generará valores muy elevados de error al elevar la diferencia al cuadrado. Se podría utilizar otro valor para el exponente de la función para evitar valores demasiado altos. También hay que tener en cuenta que los valores entre 0 y 1 cuentan con un valor de pérdida menor al

de la simple diferencia por elevarlo al cuadrado. Dependiendo del problema, puede que se necesite utilizar otro tipo de función de pérdida, que no sea una modificación de ninguna de las anteriores. No existe limitación en cuanto al tipo de función que se puede utilizar, solo debe ser derivable para que el algoritmo *backpropagation* pueda calcular las derivadas de los pesos.

- **Parámetros para las técnicas de control del entrenamiento.** En la Sección 2.8 se estudiarán algunas técnicas para controlar los principales problemas que surgen durante el entrenamiento de una red. Además de tener que elegir cuáles se van a usar, la mayoría de estas técnicas suelen tener algún parámetro que habrá que definir antes de empezar el entrenamiento.

Cuando se tenga definido el valor de los hiperparámetros se podrá realizar un experimento y, en base a las métricas alcanzadas, se decidirá cuál será la siguiente prueba a realizar.

En la siguiente sección se intentan dar algunas pautas a seguir para lograr un entrenamiento exitoso y detectar los problemas que puedan aparecer durante el entrenamiento, lo que ayudará a elegir mejor los hiperparámetros y saber cuál de ellos modificar y cómo.

## 2.8. Control y seguimiento del entrenamiento

Durante el entrenamiento de una red neuronal puede aparecer cualquiera de los problemas que se han estudiado. Sin un control y seguimiento del entrenamiento no se tendrán garantías de que este transcurra con normalidad.

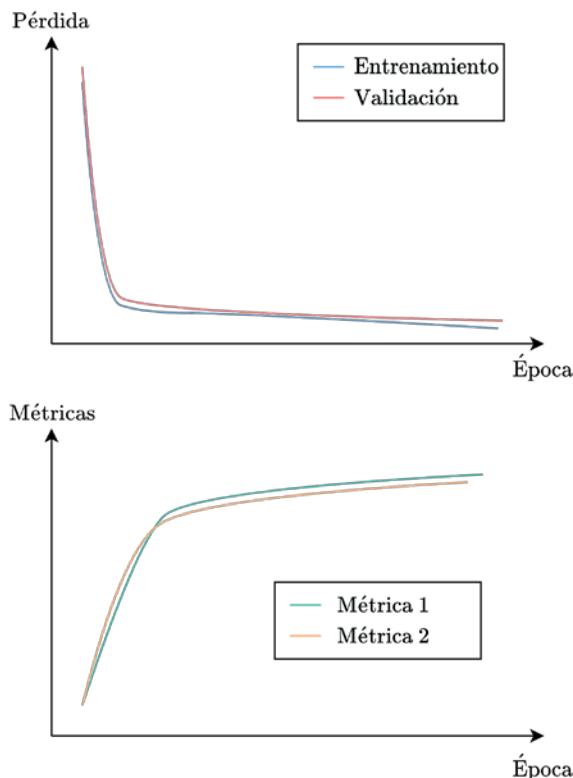
Mediante el seguimiento del entrenamiento se puede detectar cualquier error que se produzca. Lo normal suele ser llevar un registro de los valores de pérdida y de las métricas para ir estudiando su evolución durante el entrenamiento. En concreto, se suelen almacenar los valores de pérdida en cada iteración de entrenamiento (y de las métricas si se calculan en cada iteración) así como la media al finalizar cada época junto con los valores medios de pérdida y de las métricas para el conjunto de

validación. Se espera que, con el avance del entrenamiento, los valores de pérdida comiencen a reducirse y las métricas a aumentar; y que los valores obtenidos para el conjunto de entrenamiento sean parejos a los del conjunto de validación. Cualquier anomalía respecto al comportamiento descrito anteriormente suele ser síntoma de algún problema que se tendrá que corregir.

El seguimiento de los valores de pérdida y métricas se suele hacer mediante una representación gráfica donde el eje  $X$  representa las iteraciones/épocas de entrenamiento y el eje  $Y$  el valor de pérdida o las métricas según sea lo que se esté representando. En la Figura 2.35 se muestran las gráficas típicas de un entrenamiento para los valores de pérdida y métricas en los conjuntos de entrenamiento y validación.

En las gráficas de la Figura 2.35 se representan el valor medio de pérdida para las iteraciones de cada época del entrenamiento y el valor medio de las métricas para el conjunto de validación por cada época. Se observa cómo el valor de pérdida comienza siendo bastante alto y, al cabo de un cierto número de épocas, se reduce de forma notoria. En el caso de las métricas, se ve que comienzan teniendo un valor bastante bajo y acaban subiendo conforme baja el valor de pérdida. Este comportamiento se debe a que, al principio del entrenamiento, los pesos de la red son los obtenidos mediante la inicialización escogida, que suelen generar un valor alto de pérdida, pero, con el avance de las épocas de entrenamiento, el algoritmo de optimización es capaz de bajar rápidamente el valor de pérdida y, por tanto, aumentar el valor de las métricas también. Se espera que la reducción del valor de pérdida esté correlacionada con la mejora de las métricas.

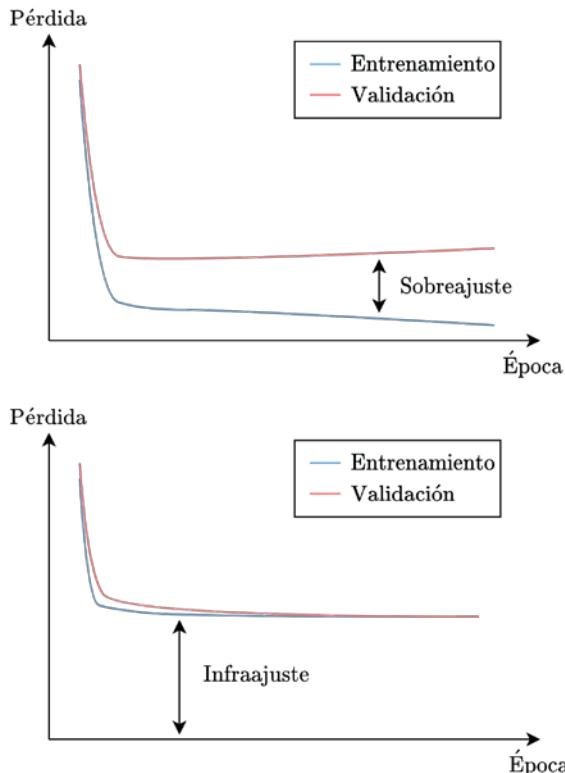
Mediante el estudio de las gráficas de pérdidas y métricas, se podrán detectar la mayoría de los problemas que se conocen, como el sobreajuste o infraajuste. Si se observa un valor bajo de pérdida para el conjunto de entrenamiento, pero alto para el conjunto de validación, significa que el modelo se está sobreajustando a los datos del conjunto de entrenamiento y no es capaz de generalizar, por lo que comete errores sobre el conjunto de validación. Por otro lado, si ambos errores van a la par, pero se quedan en unos valores altos, es síntoma de infraajuste. El modelo no cuenta con la capacidad suficiente para resolver el problema. En el primer caso se debe aumentar el conjunto de datos para que no le sea tan fácil aprendérselo, reducir la complejidad del modelo eliminando algunas capas o reducir el número de neuronas. En el segundo caso, se debe aumentar el número de capas y/o neuronas



**Figura 2.35:** Gráficas habituales del entrenamiento de una red neuronal

para que sea capaz de seguir reduciendo el error y no quedarse en valores altos. Si se dispone de métricas del conjunto de entrenamiento, también se detectan estos dos problemas analizando las métricas, aunque es suficiente con observar los valores de pérdida. En la Figura 2.36 se muestra cómo se detectan estos dos problemas en la gráfica de pérdida.

Además de los problemas inherentes al entrenamiento de estos modelos, el estudio de estas gráficas permite detectar errores que se hayan cometido durante la programación. Por ejemplo, pueden aparecer valores «*NaN*» (*not a number*) como valor

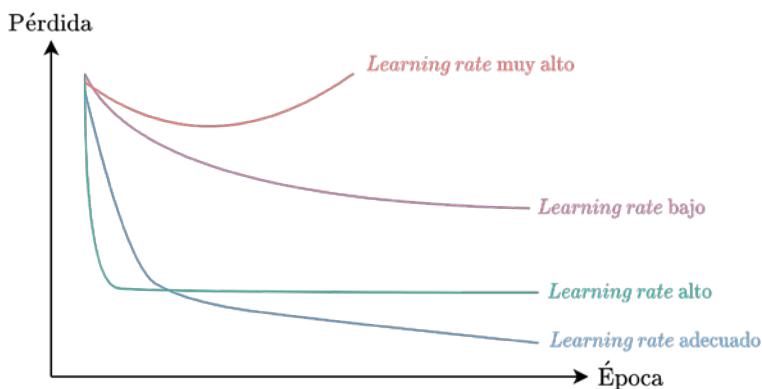


**Figura 2.36:** Detección de sobreajuste e infraajuste en la gráfica de pérdida

de pérdida, que significan que se ha intentado hacer algún cálculo que no es posible realizar, como una división por cero, o «inf» (infinito) para resultados que dan un valor infinito por multiplicar números muy grandes. En ambos casos suelen indicar que hay algún error en el código o en la elección de los hiperparámetros. Lo mismo ocurre si se observan algunos valores característicos del valor de pérdida y este no varía en todo el entrenamiento o el de alguna métrica. Si alguna vez el lector ha entrenado una red neuronal para un problema de clasificación binaria con la función *cross-entropy* y ha tenido algún error en el código puede que haya

visto aparecer el valor 0.69 como valor de pérdida. Este valor se genera cuando el vector de entrada a la función de pérdida tiene en todas sus componentes el mismo valor, normalmente 0. Por lo tanto, al aplicarle la función *softmax*, todos los componentes tienen el mismo valor de probabilidad, 0.5, y, por lo tanto, el valor de pérdida es  $-\ln(0.5) = 0.69$ . Este problema suele ser debido a que en algún momento la entrada de alguna capa o de la propia red es completamente cero. Por otro lado, analizando las métricas, si se trabaja en problema de clasificación de dos clases con una distribución en el conjunto de validación de 80-20 % para cada una de las clases respectivamente y se observa que el *accuracy* está en 80 % o 20 %, suele significar que la red está clasificando todas las entradas como una de las dos clases siempre.

La gráfica de pérdida también ayuda a elegir un buen valor para el *learning rate*. En función del valor escogido se verá descender o aumentar el valor de pérdida. El efecto del *learning rate* en la gráfica de pérdida se muestra en la Figura 2.37.



**Figura 2.37:** Efecto del valor de *learning rate* en la gráfica de pérdida

Como se observa en la Figura 2.37, un valor de *learning rate* muy alto acaba provocando que el valor de pérdida comience a crecer de manera descontrolada, al dar el paso más grande de lo necesario, acabando fuera de la zona que está indicando

el gradiente. Si el valor no es muy alto, se observará cómo el valor de pérdida baja rápidamente, pero, llegado a un punto, no es capaz de descender más al tener un paso mucho mayor al necesario. Por otro lado, un valor bajo hace que se necesiten muchas más iteraciones que las habituales para alcanzar el mínimo y apenas baje el valor de pérdida. Solo la elección de un valor de *learning rate* adecuado hará que el valor de pérdida pueda bajar paulatinamente.

No es necesario esperar varias épocas de entrenamiento para detectar algunos de estos problemas. Antes de lanzar un experimento, se puede tomar un conjunto reducido de datos o un único lote de ejemplos y ver si el modelo es capaz de sobreajustarse a él. Si no es capaz de ajustarse a un único lote de ejemplos es que hay que aumentar la complejidad del modelo para que pueda abordar el problema o algunos de los hiperparámetros del entrenamiento no tienen el valor adecuado. Cuando el entrenamiento sobre un único lote se haga correctamente y se produzca el sobreajuste casi de inmediato, se podrá pasar a entrenar con el conjunto de entrenamiento completo.

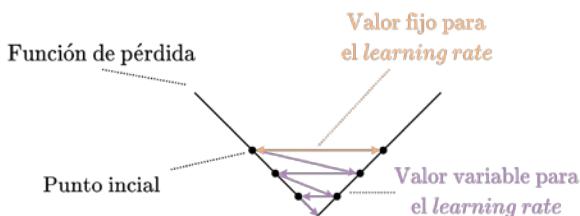
Durante el entrenamiento es habitual elegir los ejemplos de un lote aleatoriamente del conjunto de datos, de modo que no se repitan en una misma época. Al comienzo del entrenamiento se pueden ordenar de forma aleatoria para estar seguros de que no siempre se eligen los mismos. Esto ayudará a que la superficie de pérdida vaya cambiando aún más y se logre escapar de mínimos locales. De hecho, se puede llevar un registro del valor de pérdida de la red para cada ejemplo que ha estado en un lote y tenerlo en cuenta en la siguiente época para repetir ejemplos con mayor probabilidad según el valor de pérdida. De esta forma se obliga a la red a reducir el valor de pérdida en estos ejemplos difíciles.

Una vez se haya detectado algún problema se podrán tomar medidas para corregirlo. A continuación, se detallan algunas de las técnicas más utilizadas para corregir los problemas de convergencia durante el entrenamiento:

- **Equilibrar el número de ejemplos de cada clase.** Siempre que se produce sobreajuste no es debido a un modelo demasiado grande para el problema. Puede deberse al simple hecho de que una de las clases del problema sea la más predominante en el conjunto de datos. Imagine que en un problema con dos clases, el 95 % de los datos son de la primera clase. En este caso, el modelo podría optar por clasificar todas las entradas como pertenecientes a la primera clase y solo cometería un error del 5 %. Para evitar este problema se suele utilizar el mismo

número de ejemplos en cada clase para la validación y así estar seguros de que las métricas son representativas. Por otro lado, para evitar que el modelo se sobreajuste, se intenta equilibrar el número de ejemplos de cada clase en cada lote del entrenamiento. Mediante un muestreo de la clase predominante se puede obtener el mismo número de ejemplos de cada clase para entrenar.

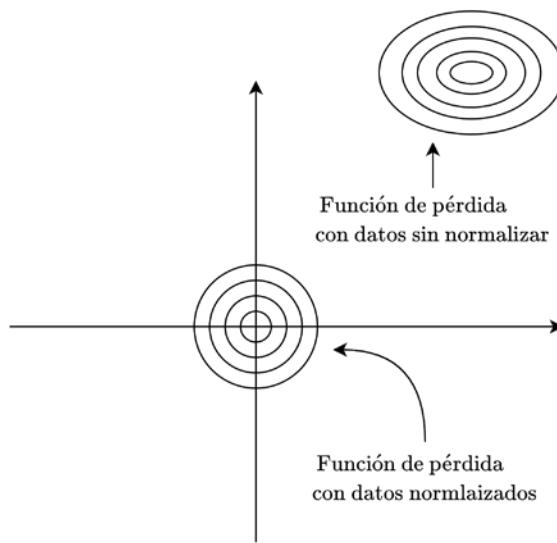
- **Políticas de actualización del *learning rate*.** En los algoritmos que no utilizan un *learning rate* adaptativo, como el descenso por gradiente, es común utilizar alguna política de actualización del *learning rate* para que no tenga siempre el mismo valor. Al comienzo, un valor alto para el *learning rate* garantiza que se descenderá rápidamente hacia el mínimo, pero puede llegar a una región donde un valor tan alto le haga imposible continuar descendiendo (ver Figura 2.38). En este instante, interesa modificar el valor del *learning rate*. Existen distintas alternativas para su actualización, la más sencilla consiste en disminuir el valor al cabo de cierto número de épocas o cuando se detecte que el valor de pérdida se ha *aplanado* y lleva varias épocas con un valor similar.



**Figura 2.38:** Diferencias en el descenso por gradiente con un valor para el *learning rate* fijo y variable

- **Normalización de los datos.** Si los datos presentan una media igual a cero y una desviación típica igual a la unidad, el entrenamiento de la red neuronal será mucho más rápido. En caso contrario, si alguna de las dimensiones de entrada se mueve en un rango mucho mayor al de las demás, provocará que la superficie de pérdida se alargue en esta dimensión mucho más que en las demás y, por lo tanto, puede provocar la aparición de oscilaciones durante el entrenamiento. En la Figura 2.39 se muestra el efecto de la normalización de los datos en la

superficie de la función de pérdida en dos dimensiones. Al mantener los valores en un rango reducido, el gradiente no tendrá un valor demasiado alto y el entrenamiento será más rápido. Además de dividir por la desviación típica y restar la media para que los valores estén en el rango  $[-1, 1]$ , también se puede normalizar dividiendo entre el máximo para dejar los valores en el rango  $[0, 1]$ .

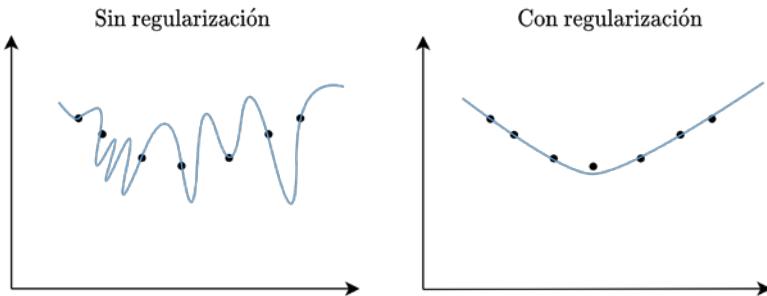


**Figura 2.39:** Superficie de la función de pérdida con datos normalizados y sin normalizar

- **Regularización.** Esta técnica consiste en añadir una penalización en el cálculo del error en base al valor de los pesos. Esto se fundamenta en el hecho de que los modelos complejos (y más propensos a tener sobreajuste) suelen contar con valores altos para sus pesos. Al utilizarse en las operaciones, cambios pequeños en la entrada se traducen en cambios bruscos en la salida. Es aquí donde la regularización logra reducir la complejidad del modelo al obligarle a utilizar pesos con valores bajos para que la función que describen los pesos del modelo tenga una forma menos compleja (ver Figura 2.40). Normalmente, se suele emplear la norma  $l_2$ , que es un término que se añade al problema de optimización y es calculada para todos los pesos,  $W$ , del modelo:

$$\arg \min_W \mathcal{L}_W + \lambda \sum \|W\|^2$$

En este caso  $\lambda$  es el parámetro de regularización y se trata de un hiperparámetro más que hay que ajustar.



**Figura 2.40:** Efecto de la regularización en la complejidad del modelo

- **Batch normalization.** Además de normalizar los datos de entrada a la red también se pueden normalizar las entradas de las distintas capas de la red. Esto es lo que hace *batch normalization*. A través de las diferentes capas de la red, los datos van sufriendo modificaciones y estos pueden acabar con valores muy altos y, en definitiva, con los problemas que tiene cualquier conjunto de datos no normalizado. *Batch normalization* propone hacer una normalización dividiendo por la desviación típica del lote actual de ejemplos y restando su media a cada dato. Además, cuenta con dos parámetros,  $\gamma$  y  $\beta$ , para realizar una transformación lineal a los datos una vez normalizados. Estos dos parámetros deben ser ajustados por la red como el resto de los pesos de las neuronas, lo que permite a cada capa ajustar los datos al rango de valores que mejor les ayude a reducir el error. Esto se traduce en un aprendizaje más rápido. Durante la etapa de entrenamiento se mantiene una media móvil exponencial para la media y desviación típica de cada lote, que luego son empleadas en la etapa de inferencia. En concreto, *batch normalization*, recibe como entrada un lote de ejemplos,  $\mathcal{B} = \{x_1, \dots, x_n\}$ , sobre los que se calcula la media y la desviación típica:

$$\begin{aligned}\mu_{\mathcal{B}} &= \frac{1}{n} \sum_{i=1}^n x_i \\ \sigma_{\mathcal{B}}^2 &= \frac{1}{n} \sum_{i=1}^n (x_i - \mu_{\mathcal{B}})^2\end{aligned}$$

Estos dos valores son utilizados para normalizar los datos:

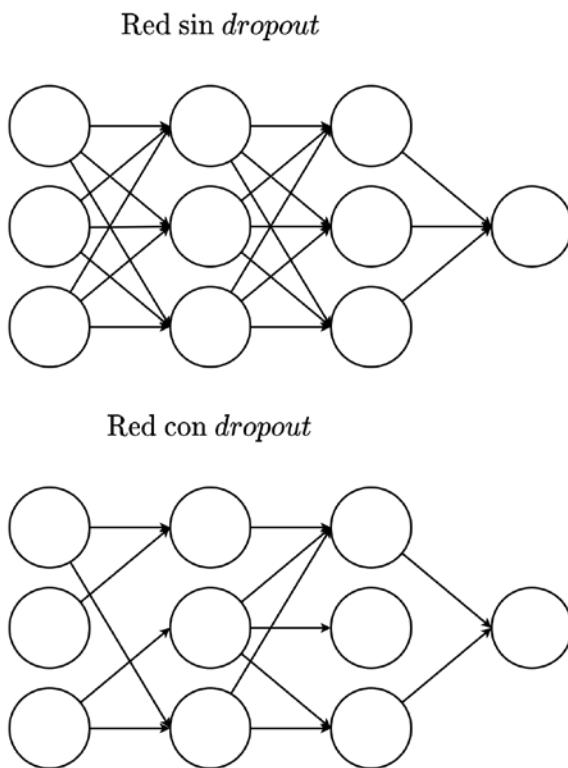
$$x'_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

donde  $\epsilon$  es un valor muy pequeño para evitar la división por cero. Por último, el lote de ejemplos de salida,  $\mathcal{B}' = \{y_1, \dots, y_n\}$ , se obtiene realizando una transformación lineal,  $y$ , a los datos que han sido normalizados,  $x'$ , según los valores  $\gamma$  y  $\beta$  (aprendidos durante el entrenamiento):

$$\begin{aligned}y_i &= \gamma x'_i + \beta \\ \mathcal{B}' &= \{y_1, \dots, y_n\}\end{aligned}$$

- **Dropout.** Esta técnica busca evitar el sobreajuste convirtiendo a la red neuronal en un conjunto de redes más simples que aprenden a resolver el problema promediando sus salidas. Para lograrlo, aleatoriamente, se «desactiva» de la red algunas de las neuronas de la capa en la que se aplique esta técnica durante la propagación hacia delante (ver Figura 2.41). Esto implica que, si hay  $n$  neuronas en la capa, al desactivar algunas aleatoriamente, se obtiene hasta  $2^n$  redes más pequeñas. Para cada iteración del entrenamiento, se escoge una de estas posibles redes y se entrena. De forma que, entrenar una red con esta técnica puede verse como entrenar una colección de  $2^n$  redes más pequeñas. En la etapa de inferencia, no se utiliza el *dropout*, se utiliza la red completa, lo que provoca que se obtenga el resultado de la red como una ponderación de las diferentes subredes que fueron entrenadas. Gracias a esto, se puede controlar mejor el sobreajuste, al provocar que exista una redundancia en la red. La desactivación de las neuronas se controla mediante un parámetro,  $p$ , que indica la probabilidad para cada neurona de acabar desactivada. La desactivación de una neurona consiste en poner su salida a cero. En aquellas neuronas donde no se aplica *dropout*,

su salida es escalada por  $1/(1-p)$ , de forma que la suma sobre todas las entradas permanezca inalterada.



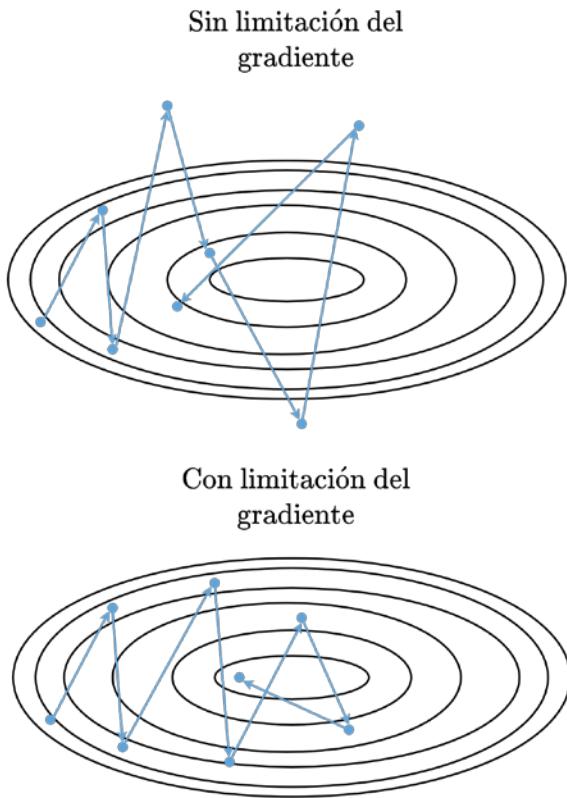
**Figura 2.41:** Red con y sin *dropout*

- **Aumento de los datos.** Una alternativa a modificar la arquitectura de la red o a utilizar técnicas para controlar el sobreajuste es el aumento del conjunto de datos. Imagine que se dispone de un conjunto de imágenes médicas pertenecientes a radiografías de un conjunto muy limitado de pacientes. Lo más probable es que la red acabe sobreajustándose a los datos, al no haber suficientes para que pueda generalizar. En estos casos, se puede optar por aumentar artificialmente

el conjunto de datos. En el ejemplo que se ha puesto, basta con girar, introducir ruido, recortar, cambiar brillo y contraste, etc. Las técnicas de aumento de datos serán específicas del problema que se esté tratando y habrá que consultar la literatura para encontrar ejemplos de técnicas con resultados contrastados. El objetivo es generar datos que sean diferentes del dato original y que se pueda seguir identificando como de la clase que tienen asociada originalmente.

- **Limitación del gradiente.** No siempre es buena idea utilizar el valor que del gradiente sin ningún tipo de control. Algunas funciones de pérdida como ECM generan gradientes bastante grandes si aparecen *outliers* en los datos o, si estos están por normalizar, puede tener gradientes demasiado grandes por los valores que se utilizan. En esta situación, si se intenta actualizar los pesos con estos gradientes, acabarán situados en una zona alejadas del mínimo, donde el valor de los pesos o de pérdida pueden llegar a ser lo suficiente grandes como para valer infinito. La limitación del gradiente consiste en poner un valor máximo (y mínimo) para el gradiente, de forma que nunca pueda superarlo y estar seguros de que nunca se desplaza más de una cierta cantidad en una única iteración de entrenamiento. En la Figura 2.42 se muestra el efecto de limitar el gradiente cuando la superficie de pérdida produce oscilaciones.

Una vez se lance el entrenamiento, el trabajo consistirá en hacer un seguimiento de este mediante los registros y gráficas que se hayan programados para detectar posibles problemas o cambios que se deberían hacer en el siguiente experimento y lograr así mejores resultados.



**Figura 2.42:** Efecto de la limitación del gradiente durante el entrenamiento



## Capítulo 3

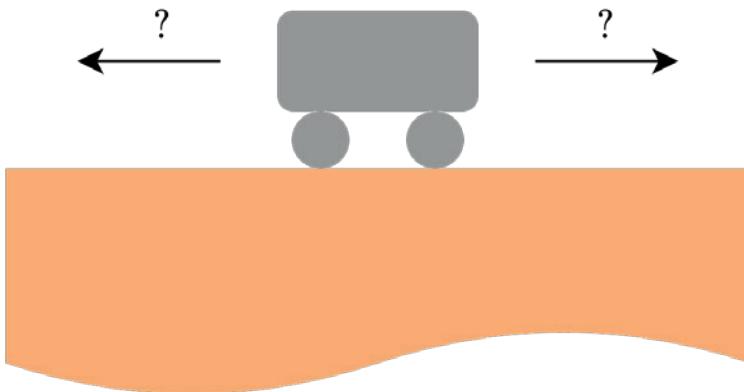
# Redes neuronales recurrentes

### 3.1. Introducción

Existen algunos tipos de datos que tienen una componente temporal inherente. Cuando se lee un texto o se habla está presente un orden temporal en los datos que el cerebro procesa. Cada imagen (en el caso de la lectura) o sonido (en el caso del habla) llega uno detrás de otro, permitiendo reconocer letras que luego se convierten en un enunciado.

En estos casos, cada dato individual carece de sentido (imagen o sonido aislado) y lo que interesa es la **secuencia** completa. No es lo mismo la frase «Javier le ha comprado el coche a Juan» que la frase «Juan le ha comprado el coche a Javier». Si se descompone la frase anterior en cada uno de sus componentes (palabras) y se analizan individualmente, no se sabe de quién es el coche ni quién es el que lo compra. Solo la creación de una secuencia que dote de orden a las palabras de la frase anterior puede responder estas preguntas.

Hay problemas que precisan del uso de secuencias para ser resueltos, se necesita saber el contexto para poder abordarlos. Por ejemplo, dada la imagen de la Figura 3.1, no se conoce cuál será la posición del vagón en el siguiente fotograma. Sin embargo, si se dispone de un vídeo con los fotogramas anteriores al actual, se podrá predecir con mayor seguridad la posición (ver Figura 3.2). La única forma de resolver este problema es disponer de una secuencia de imágenes.

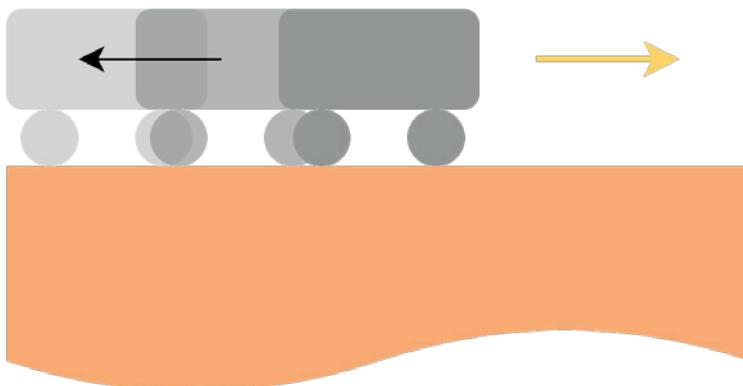


**Figura 3.1:** Sin información de los instantes anteriores, no se puede predecir la siguiente posición del vagón

El cerebro está constantemente analizando secuencias de datos como imágenes y sonido. La componente temporal juega un papel esencial en la forma en la que el cerebro procesa la información. Gracias al uso de la memoria, el cerebro es capaz de almacenar información sensorial del pasado que le ayuda a resolver problemas en el presente.

En la versión digital, las redes neuronales artificiales no cuentan con un mecanismo explícito que les permita procesar secuencias de datos. Las redes neuronales no tienen **memoria**. Por lo tanto, los problemas relacionados con el procesamiento de secuencias tienen que ser tratados como un problema típico de redes neuronales. Por ejemplo, si se quiere adivinar la siguiente palabra de una frase con una red neuronal, se puede optar por alguna de las siguientes alternativas:

- **Toda la frase como entrada.** Al utilizar toda la frase como entrada de la red, se establece el tamaño de la entrada y, por lo tanto, de la frase a un valor fijo. No se podrá analizar frases más grandes que el tamaño que se fije. Además, si el tamaño fijado para la entrada es demasiado grande, se puede acabar con



**Figura 3.2:** En base a la información anterior, se puede predecir que el vagón se moverá hacia la derecha

millones de pesos por cada neurona, puesto que cada una tiene un peso para cada componente del vector de entrada.

- **Una ventana de tamaño fijo.** La alternativa a la opción anterior consiste en darle a la red una entrada de tamaño fijo pero compuesta por solo algunas de las palabras finales. De esta forma, se puede procesar cualquier frase, por larga que sea, y reducir el número de parámetros respecto a la alternativa anterior. No obstante, algunas predicciones pueden tener dependencias a largo plazo que no estén presentes en la ventana de entrada a la red. Por ejemplo, si se utiliza las dos palabras anteriores como entrada de la red para la frase «Madrid, que ciudad tan bonita, me encantaría volver a», la entrada consistirá en un vector (en codificación *one-hot*, por ejemplo) con las palabras «volver a». Sin embargo, es en el comienzo de la frase donde está la información que se necesita para predecir correctamente la siguiente palabra.

Independientemente de la alternativa que se utilice, otra desventaja del uso de las redes neuronales para analizar secuencia es el hecho de que las entradas son independientes unas de otras. La red define unos pesos para cada componente de la entrada, de forma que, para las entradas «Me gusta este coche» y «Este coche me

gusta», los pesos que procesan en la primera frase «coche» no son los mismos que en la segunda, donde «coche» aparece más al comienzo y son otros pesos los que procesan «coche». La red neuronal no es invariante a la posición, cualquier cambio en el orden de las palabras puede generar una salida completamente diferente, aunque ambas entradas signifiquen lo mismo. No se puede reutilizar los pesos que permiten detectar «coche» en la primera frase para detectarlo en la segunda, hay que definir unos pesos para que detecten «coche» en una posición específica. Esta explosión combinatoria de palabras y posiciones dentro de la frase da una idea de por qué es muy difícil, por no decir imposible, procesar secuencias con redes neuronales tradicionales.

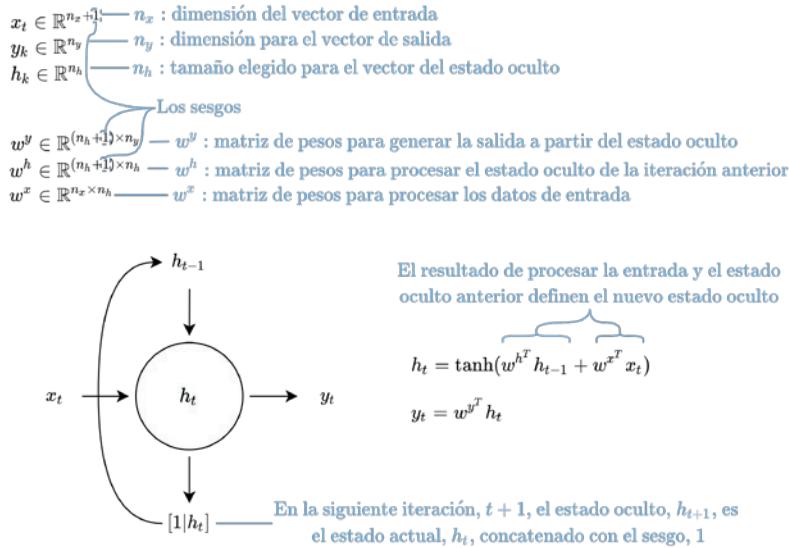
Las redes neuronales se encuentran limitadas para analizar secuencias, no son el modelo más indicado. Para este propósito se crearon las redes neuronales recurrentes.

## 3.2. Redes neuronales recurrentes

Las redes neuronales que se estudiaron en el Capítulo 1 son conocidas como redes *feedforward*. Reciben una entrada, esta es procesada por la red y genera una salida. Funcionan en un único sentido, es decir, los componentes de la red no presentan ciclos. La red es un grafo acíclico dirigido.

Al contrario que en las redes *feedforward*, las **redes neuronales recurrentes** o redes recurrentes, *recurrent neural networks* (RNN) en inglés, llamadas así por primera vez por John Hopfield [17] (aunque se venían usando desde mucho antes), son redes neuronales tradicionales que cuentan con una retroalimentación que les permite tener una memoria o **estado oculto** (*hidden state* en inglés) con información del contexto de la entrada actual. Esta retroalimentación se consigue gracias a una conexión en bucle que transmite el estado oculto de la iteración anterior como entrada de la iteración actual. Los datos de entrada junto con el estado oculto de la iteración anterior son utilizados para calcular el estado oculto actual. En lugar de calcular la salida de la red en base a los datos de entrada, las redes recurrentes calculan la salida en base a su estado oculto, que tiene en cuenta tanto los datos de

entrada como el estado oculto de la iteración anterior. En la Figura 3.3 se muestra el esquema de una red recurrente.



**Figura 3.3:** Esquema de una red neuronal recurrente

Tal y como se observa en Figura 3.3, para facilitar la comprensión, todas las neuronas de una capa se representan como una única neurona cuyos pesos engloban el procesamiento que realiza cada neurona de la capa. Esta neurona se centra en calcular su estado oculto,  $h$ , en cada nueva iteración,  $t$ , a partir de los datos de entrada en la iteración actual,  $x_t$ , y el estado oculto de la iteración anterior  $h_{t-1}$ . La salida actual de la red,  $y_t$ , no es más que una transformación lineal del estado oculto actual,  $h_t$ . Al igual que en la versión tradicional, la función de activación elegida para la neurona no se considera parte de esta y se aplica a cada componente del vector de salida de forma individual.

Los cálculos que se realizan en la red recurrente son similares a los que se hacen en una red neuronal tradicional. La combinación lineal del estado oculto,  $h_t$ , y la matriz de pesos,  $w^y$ , generan la salida de la red,  $y$ . Por otro lado, la combinación

lineal de las entradas actuales,  $x_t$ , con el vector de pesos,  $w^x$ , junto con la combinación lineal del estado oculto de la iteración anterior,  $h_{t-1}$ , con la matriz de pesos,  $w^h$ , permiten calcular el estado oculto de la iteración actual,  $h_t$ , mediante la suma de las dos operaciones anteriores y la aplicación de la función de activación tanh. La elección de esta función de activación se debe a que, además de introducir no linealidades en el estado oculto, controla mejor el problema del desvanecimiento del gradiente (muy habitual en estas redes) que otras funciones de activación. Al tener la salida en el rango  $[-1, 1]$ , es menos probable que aparezcan valores muy altos o bajos puesto que, en la siguiente iteración, se recibe como entrada el estado oculto actual cuyos valores positivos o negativos pueden incrementar o reducir respectivamente el valor del estado oculto siguiente, lo que permite controlar el crecimiento de los valores del estado oculto.

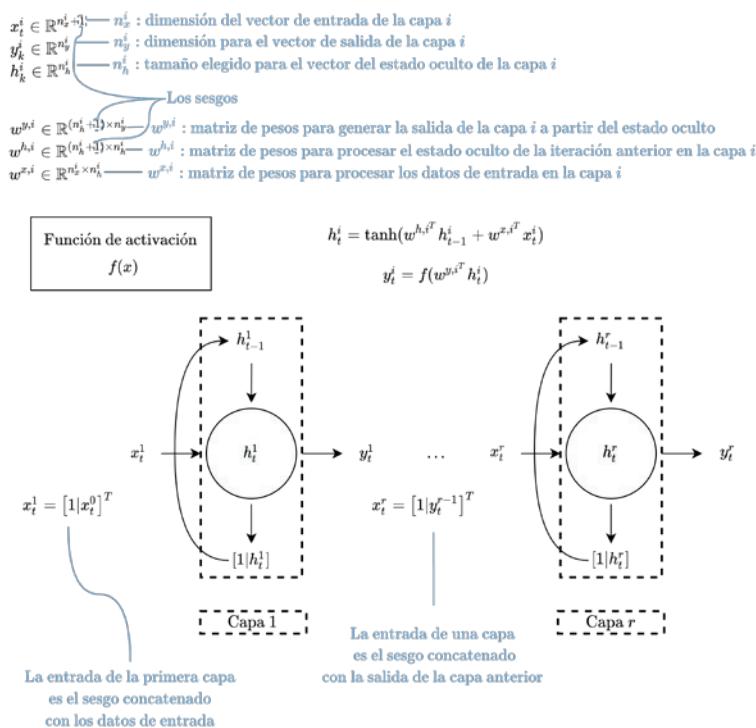
La dimensión de la entrada,  $x$ , así como de la salida,  $y$ , vendrán determinadas por los datos del problema. Sin embargo, la dimensión del estado oculto,  $h$ , debe decidirse antes de entrenar y será un hiperparámetro más que habrá que ajustar durante el entrenamiento de la red en función del problema que se esté abordando.

Otro aspecto importante de estas redes es que, independientemente del número de iteraciones que se realicen, siempre se utilizan los mismos pesos en todas las iteraciones, no hay que utilizar más pesos si la entrada es más grande como ocurría en las redes neuronales tradicionales. Si se analiza una frase carácter a carácter, todos van a ser procesados por los mismos pesos solo que, en cada instante, el estado oculto irá cambiando en función de los pesos aprendidos por la red para el problema y permitirá dar una salida diferente para una misma entrada en base al contexto actual de la entrada, que viene dado por el estado oculto.

La inicialización de los pesos de la red se hace con las mismas técnicas que se vieron para las redes neuronales en la Sección 2.6.1. Además de los pesos, hay que darle un valor inicial al estado oculto,  $h$ , que será el que se utilizará como estado oculto anterior a la primera iteración. Existen diferentes alternativas para inicializar el estado oculto, la más sencilla consiste en inicializarlo a un valor fijo, por ejemplo 0. Esta opción no es la más recomendable, ya que no produce buenos resultados y puede llegar a generar sobreajuste. Otra alternativa podría ser inicializar el estado oculto con algún valor aleatorio como los pesos, pero, en lugar de esto, se puede utilizar unos pesos entrenables como valores iniciales del estado oculto. De esta forma, el estado oculto inicial será un parámetro más que deberá ajustar la red.

En base a los datos del problema, la red puede aprender un buen estado oculto de partida que le permita alcanzar mejores resultados que una simple inicialización aleatoria o constante. No obstante, sin un conjunto de datos de entrenamiento lo suficientemente grande, la red no logrará aprender un buen estado inicial y puede que una simple inicialización aleatoria logre mejores resultados.

Para formar una red recurrente multicapa se procede como en las redes neuronales. Cada neurona recurrente forma una capa en sí, por lo que solo hay que añadir tantas neuronas como capas se deseen y conectar la salida de cada una como entrada de la siguiente. La salida de la última neurona es la salida de la red (ver Figura 3.4).



**Figura 3.4:** Esquema de una red neuronal recurrente multicapa

Cada neurona de la red tiene su propio estado oculto y su matriz de pesos independiente de las otras. Como en la red neuronal, interesa obtener la salida de la red, es decir, la salida de la última capa, que se calcula procesando la entrada por la primera neurona (capa) y suministrando su salida como entrada de la siguiente hasta que se llega a obtener la salida de la última neurona (capa).

### 3.2.1. Entrenamiento de una red neuronal recurrente

El entrenamiento de las redes neuronales recurrentes es similar al realizado en las redes neuronales tradicionales. El entrenamiento de la red se plantea como un problema de optimización matemática donde se debe minimizar una función de pérdida,  $\mathcal{L}$ . Para minimizar esta función se utilizan los mismos algoritmos de optimización iterativos basados en el gradiente de la función de pérdida que se estudiaron en la Sección 2.6 para las redes tradicionales. En las neuronas (capas) ocultas se emplea *backpropagation* para calcular la derivada de los pesos de estas neuronas (capas).

El conjunto de datos de entrenamiento,  $\mathcal{D}$ , se compone de una secuencia de datos de entrada,  $x$ , y una secuencia de valores de salida esperada,  $y$ . Un ejemplo de entrenamiento lo compone un dato de la secuencia de entrada en un instante  $t$ ,  $x_t$ , y la salida esperada para el instante  $t$ ,  $y_t$ . A diferencia de las redes neuronales tradicionales, no se puede separar los datos de cualquier forma para crear los conjuntos de entrenamiento, validación y prueba, puesto que los datos no son independientes unos de otros y siguen un orden dentro de la secuencia. Cada dato de la secuencia cuenta con un contexto que viene definido por los datos de la secuencia en instantes anteriores y que la red captura en su estado oculto,  $h$ . Si no se respeta el orden de los datos, la red creará un contexto de la entrada actual que no tiene que ver con el real. Por ejemplo, si se intenta predecir la temperatura dentro de dos días en base a la temperatura actual, la red no puede hacer este trabajo correctamente a menos que haya recibido los datos en orden de los días anteriores para generar su estado oculto.

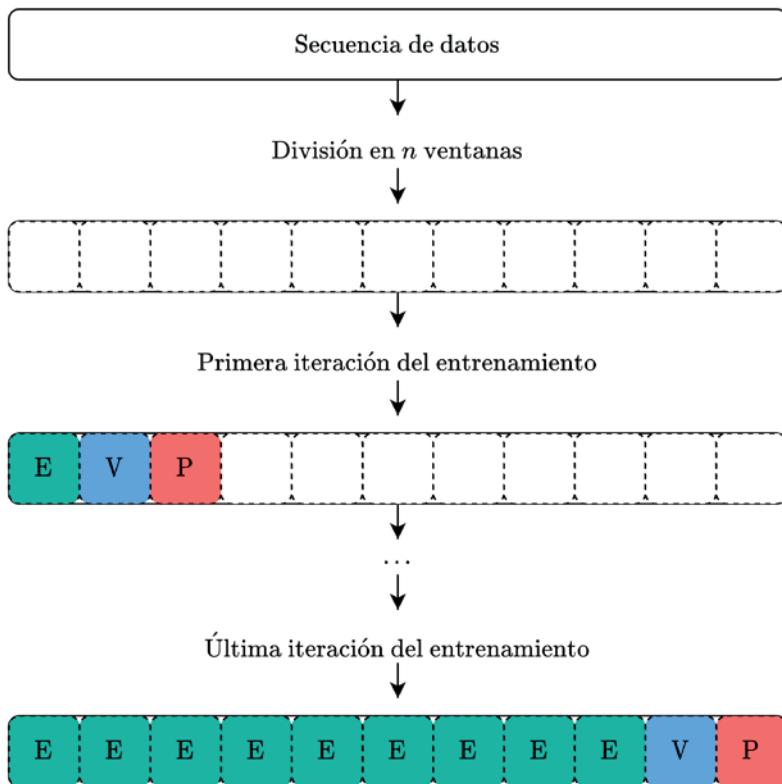
Existen diferentes alternativas a la hora de trabajar con secuencias de datos para

entrenar una red recurrente. La solución más simple consiste en dividir en tres la secuencia original, de forma que la primera parte se utiliza para entrenar la red y la segunda y tercera para validar y pruebas respectivamente. La desventaja de la solución anterior es que gran parte de los datos se quedan sin utilizar para entrenar y algunos de ellos pueden contener situaciones que no aparecen en el conjunto de entrenamiento. Otra alternativa es utilizar una ventana deslizante definida por un número de instantes de tiempo. Se divide el conjunto original en estas ventanas y se comienza utilizando solo la primera para entrenar, la siguiente para validar y la tercera para pruebas. En el siguiente instante de entrenamiento se desplazan una unidad las ventanas y se vuelve a entrenar (ver Figura 3.5). Con esta configuración, se puede aprovechar todos los datos de la secuencia para entrenar con ellos.

Cuando se tiene lista la división del conjunto de datos se puede realizar el entrenamiento de la red. Un cambio importante respecto al entrenamiento de las redes tradicionales y las redes recurrentes es el uso de la componente temporal. La salida de la red en un instante  $t$  se genera, no solo en función de la entrada actual, sino del estado oculto de la red, que a su vez se genera a partir de los datos de todos los instantes anteriores hasta el primer instante. Por lo tanto, *backpropagation* tendría que calcular la derivada de los pesos teniendo en cuenta esta situación, ya que los mismos pesos se utilizan en varios instantes de tiempo. Para entrenar las redes recurrentes se utiliza una versión modificada del algoritmo *backpropagation* conocida como *backpropagation* a través del tiempo, *backpropagation through time* (BPTT) en inglés, que define un número determinado de instantes sobre los que *backpropagation* calculará el *forward pass* para después calcular la derivada de los pesos durante el *backward pass*. Esto permite que el cálculo de las derivadas sea posible en los equipos actuales, al limitar los requerimientos de procesamiento y almacenamiento.

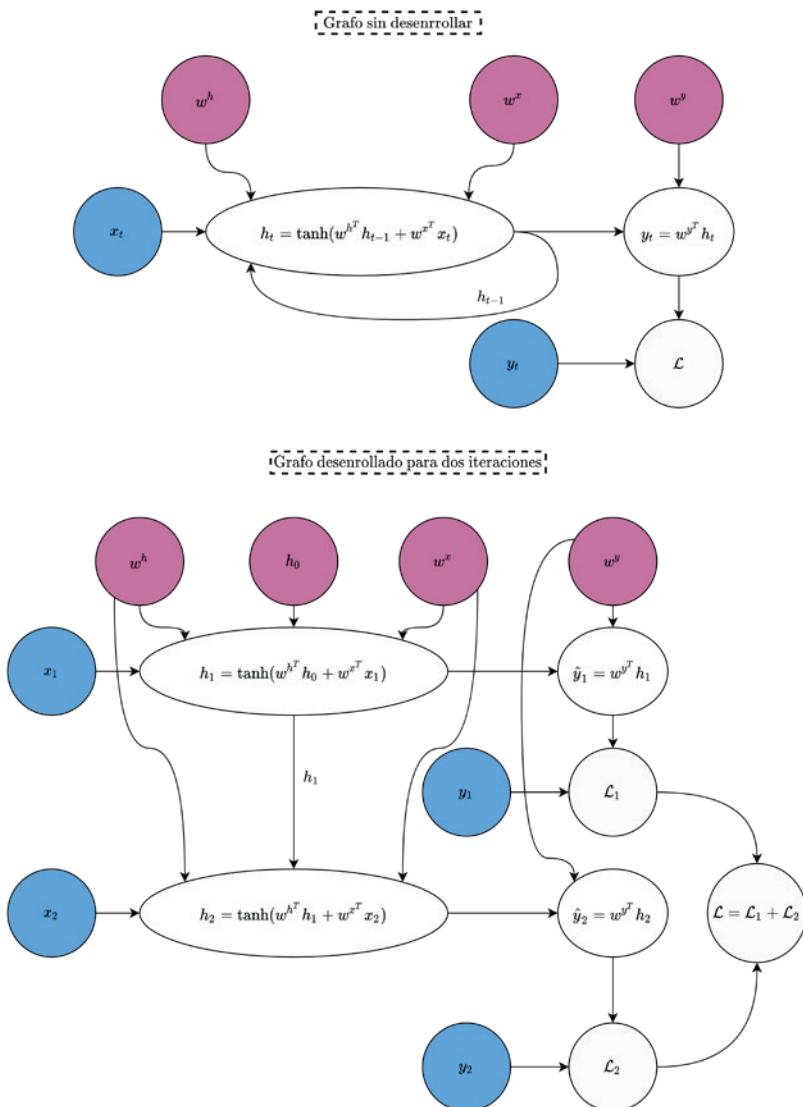
Al igual que en las redes neuronales tradicionales, en las redes recurrentes se aplica el algoritmo BPTT sobre el grafo de operaciones de la red. Este grafo para las redes recurrentes requiere de «desenrollar» las iteraciones que la red realiza en cada iteración para crear un grafo dirigido sin ciclos. En la Figura 3.6 se muestra un ejemplo de cómo sería este proceso para dos iteraciones del entrenamiento. El error final consiste en la suma de los errores en cada iteración.

Una vez se obtiene el grafo de operaciones desenrollado (sin ciclos) se pueden cal-



**Figura 3.5:** División en ventanas de la secuencia de datos y uso en cada iteración de entrenamiento para entrenar (E), validar (V) y pruebas (P)

cular las derivadas de la misma forma que se calculan en las redes neuronales tradicionales con *backpropagation*.



**Figura 3.6:** Proceso de desenrollado del grafo de operaciones para dos iteraciones de una red neuronal recurrente

### 3.2.2. Aplicaciones

Las aplicaciones de las redes neuronales recurrentes son principalmente el procesamiento de secuencias de datos. Se entiende por secuencia de datos a cualquier conjunto de datos con un orden asociado. Un texto, un audio, un video o series temporales son ejemplos de secuencias de datos.

De forma general, los problemas relacionados con datos secuenciales se conocen como **modelado de secuencias**, *sequence modeling* en inglés. Dentro del modelado de secuencias se distinguen los problemas en función del uso que se haga de las entradas y salidas de la red:

- **Muchos a muchos.** En este tipo de problemas se tiene una secuencia de entrada y la salida es otra secuencia. Este puede ser el caso de problemas como la traducción de un texto, donde la entrada es una secuencia (el texto en el idioma original) y la salida es otra (el texto en el idioma destino).
- **Muchos a uno.** Para este problema se ignora (no se calcula) las salidas en los instantes anteriores al último y solo se está interesado en la salida de la red en el último instante de tiempo, cuando ya ha procesado toda la secuencia de entrada. Un problema de este tipo podría ser el análisis de sentimientos en texto: la entrada puede ser un comentario de un cliente y la salida es la clasificación de la secuencia de entrada en comentario positivo y negativo, por ejemplo.
- **Uno a muchos.** Este problema recibe como entrada el mismo dato en todas las iteraciones y, en base al estado anterior, la salida va cambiando. Un ejemplo de este tipo de problemas es la generación de una descripción de una imagen: la entrada a la red es la imagen (siempre es la misma) y la salida son las palabras que genera la red en cada iteración.
- **Uno a uno.** Los problemas uno a uno no se consideran un problema de modelado de secuencias, puesto que ésta no existe. Son los problemas que se abordan con una red neuronal tradicional: hay un dato de entrada independiente con una única salida, que solo depende del dato de entrada actual.

Dentro de las aplicaciones de las redes neuronales recurrentes destacan la traducción, la conversión de voz en texto, la predicción de series temporales o el procesamiento del lenguaje natural, *natural language processing* (NLP) en inglés, que permite crear *chatbots*.

Las redes neuronales recurrentes pueden ser utilizadas para el modelado de sistemas

mas dinámicos para su control. No obstante, hay que tener presente que las redes neuronales recurrentes trabajan de forma discreta y no continua.

### 3.2.3. Limitaciones

Las redes neuronales recurrentes son consideradas máquinas de Turing completas [18], lo que significa que pueden ser utilizadas para modelar cualquier **algoritmo**. Sin embargo, al igual que en las redes neuronales, no se dispone de ninguna información acerca de la arquitectura correcta o el valor de los pesos para cada problema, tienen que obtenerse mediante un proceso de ensayo y error.

Otra de las limitaciones que se encuentran en estas redes es su entrenamiento: al tener que limitar *backpropagation* a un número prefijado de iteraciones, no se puede entrenar con toda la secuencia completa y puede que se esté descartando información relevante para dar la salida. A este problema hay que añadirle el desvanecimiento del gradiente que se produce al ir calculando el estado oculto en cada iteración con la función de activación tanh y que supone uno de los mayores retos de su entrenamiento. Por este motivo, las redes recurrentes también tienen problemas para acceder a la información más lejanas del instante actual, que en algunos problemas puede ser relevante.

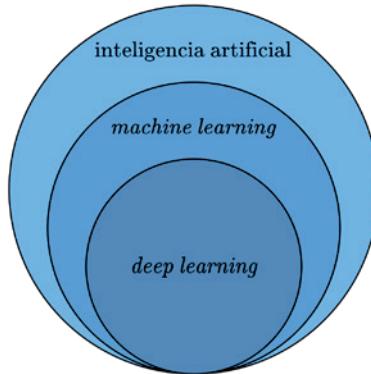


# Capítulo 4

## *Deep learning*

### 4.1. Introducción

El aprendizaje profundo, *deep learning* en inglés, es un campo del aprendizaje automático, *machine learning* en inglés, que se engloba, a su vez, dentro de la inteligencia artificial (ver Figura 4.1).



**Figura 4.1:** Relación entre inteligencia artificial, *machine learning* y *deep learning*

Cuando se trabaja con las técnicas de *machine learning* se utiliza un conjunto de datos para ajustar un modelo predefinido que resuelve un determinado problema. Los datos con los que trabaja el modelo no suelen ser los datos originales, los datos

en bruto del problema. En su lugar, se suelen definir una serie de características, *features* en inglés, que se obtienen a partir de los datos originales y son las que utiliza el modelo para resolver el problema.

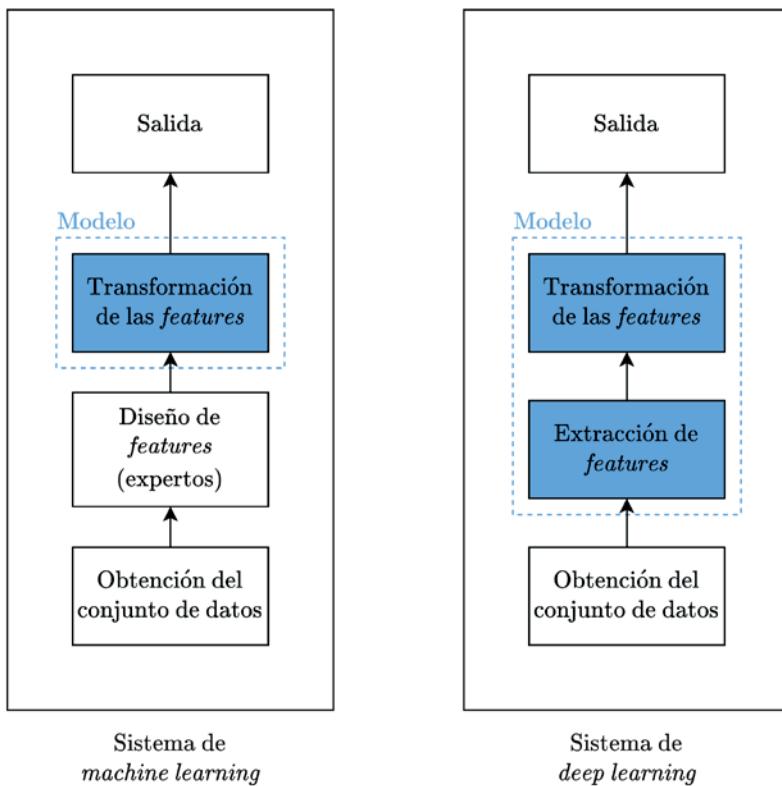
Este enfoque requiere de un estudio previo del problema por una serie de expertos que ayuden a definir las mejores características para el modelo, lo que supone una ingente cantidad de tiempo y dinero. Esta ha sido la forma tradicional de trabajar con los modelos de *machine learning*. Una vez se dispone de las características más relevantes para solucionar el problema, se elige un modelo de *machine learning* (una red neuronal p. ej.) y este se encarga de aprender la transformación que hay que aplicarle a las características elegidas para obtener la salida esperada.

En contraste con el enfoque tradicional del *machine learning*, las técnicas de *deep learning* se encargan de extraer, por sí solas, las características más relevantes de los datos originales para resolver el problema, además de aprender la transformación que hay que aplicar a estas características para dar la salida esperada.

En la Figura 4.2 se muestra una comparativa entre el desarrollo de un sistema basado en técnicas de *machine learning* y *deep learning*.

El perceptrón multicapa se considera el modelo de *deep learning* más básico. Con solo dos capas, ya se dispone un aproximador universal de funciones. Gracias a la popularización del algoritmo *backpropagation* en la década de los 80, comenzaron a aparecer redes neuronales con más de una capa, llamadas redes neuronales profundas (*deep neural networks* en inglés) para distinguirlas de las redes con una única capa, conocidas como redes poco profundas (*shallow neural networks* en inglés). Aunque, en principio, con dos capas es suficiente para aproximar cualquier función, en la práctica, el número de neuronas necesarias para resolver un problema real con una única capa oculta sería demasiado elevado. En lugar de aumentar la anchura de la capa oculta (su número de neuronas), se suele optar por aumentar la profundidad de la red añadiendo más capas. De esta forma, por lo general, el número de neuronas necesarias para resolver el problema crece de manera lineal y no exponencial como en el primer caso.

Debido a las limitaciones de una red neuronal con una única capa, en los comienzos, existía un gran interés en utilizar redes neuronales muy profundas para dotar de más capacidad a la red. A pesar de disponer del algoritmo *backpropagation*, el entrenamiento de redes neuronales profundas era bastante difícil en aquella época,



**Figura 4.2:** Comparativa entre el desarrollo de un sistema basado en técnicas de *machine learning* y *deep learning*

en parte por las limitaciones en cuanto a capacidad de procesamiento. No fue hasta el año 2006, cuando, en un trabajo publicado por Geoffrey E. Hinton, Simon Osindero y Yee-Whye Teh, se logró entrenar redes con un número de capas mucho mayor a las que se habían logrado entrenar hasta la fecha [19]. Esto motivó a otros investigadores a comenzar a entrenar redes neuronales más profundas que las habituales y, para destacar este hecho, se comenzó a utilizar el término ***deep learning***.

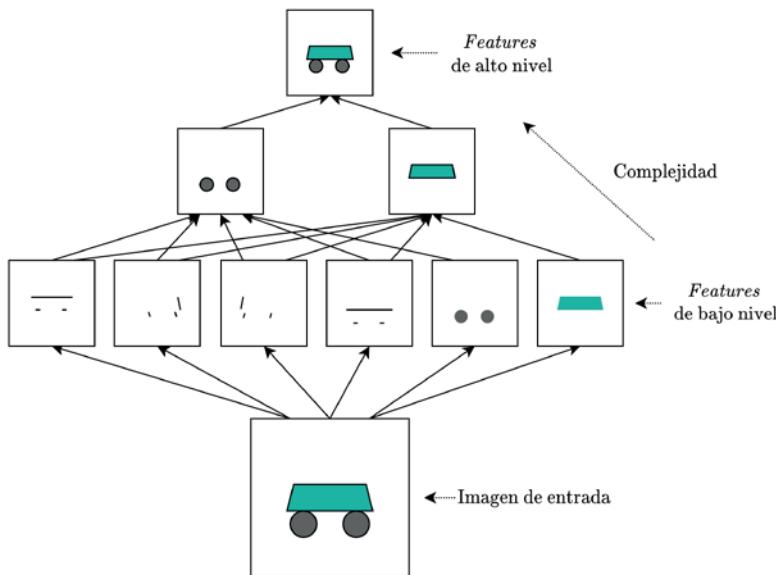
para referirse al entrenamiento de redes **más profundas** que las entrenadas hasta ese momento.

Una de las características de las técnicas de *deep learning* es que la extracción de características que realizan de los datos originales suele generar una representación jerárquica distribuida: se extraen características de menor a mayor complejidad, de forma independiente, definiéndose las características de los niveles superiores en función de las características de los niveles inferiores. El objetivo de las técnicas de *deep learning* consiste en encontrar unas características de alto nivel, de forma que el modelo pueda transformar estas características en la salida esperada, con más facilidad que empleando la representación original de los datos. Por ejemplo, en el caso de la clasificación de imágenes, las técnicas de *deep learning* procesan la imagen de entrada para, en un primer nivel, extraer características de bajo nivel (como colores y bordes), mientras que en niveles superiores se detectan características más complejas, de más alto nivel (como pueden ser las ruedas de un coche o la cara de una persona). Estas características de más alto nivel serán las utilizadas para generar la salida del modelo.

En la Figura 4.3 se observa la jerarquía de características que generaría un modelo de *deep learning* al procesar una imagen. Se observa cómo, en el nivel más alto de la jerarquía, es capaz de detectar el objeto de la entrada en base a la detección de características de más bajo nivel que se ha hecho en capas anteriores.

De forma general, en un modelo de *deep learning* se distinguen dos etapas: una **etapa de extracción** de características y una **etapa de transformación de características**. La primera de las etapas consiste en crear la jerarquía de características a partir de los datos en bruto del problema. La segunda etapa toma las características de más alto nivel de la jerarquía y las utiliza para aplicarles una transformación que permita dar la salida esperada por el sistema. Por ejemplo, en el caso de una red neuronal aplicada a la clasificación de imágenes, las primeras capas podrían realizar una extracción de características de bajo nivel y, al ir aumentando la profundidad de la red, las características extraídas serían más complejas, definidas por las características extraídas en las capas de más bajo nivel. La última capa de la red puede utilizarse para aprender a realizar una transformación de las características de más alto nivel a la salida esperada. En la Figura 4.4 se muestra el esquema completo de un modelo de *deep learning* con sus diferentes etapas.

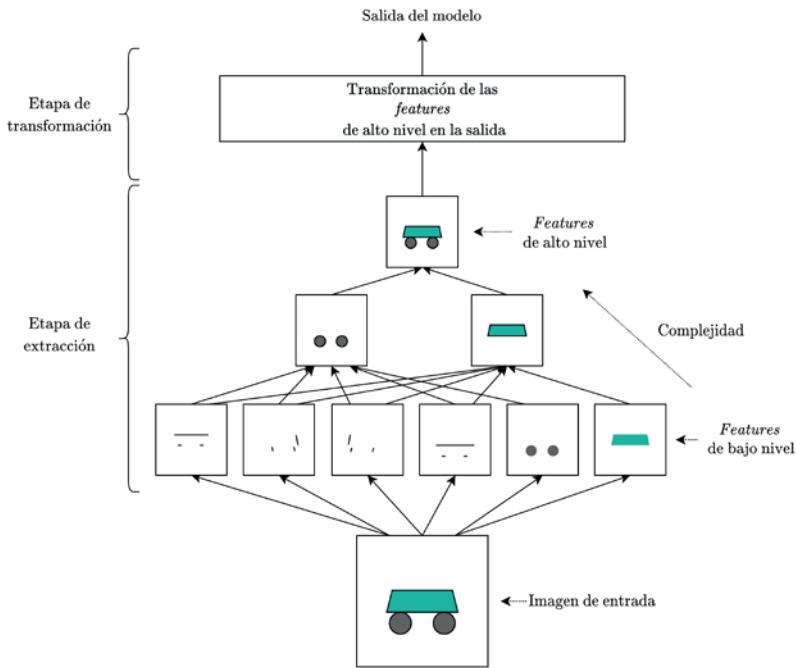
La principal desventaja de utilizar redes neuronales para la extracción de caracte-



**Figura 4.3:** Descomposición de los píxeles de una imagen en una nueva representación jerárquica de características de menor a mayor complejidad

ticas es su bajo desempeño, puesto que no son invariantes a la posición: la misma persona en una imagen, pero desplazada, será procesada por unos pesos totalmente diferentes a los que la procesaron en la posición original, además de suponer una ingente cantidad de pesos que deben ser ajustados (uno por cada posición de la imagen y neurona de la red). Esto limita en gran medida la capacidad de las redes neuronales para trabajar extrayendo características y es por lo que se definen características a mano (nivel de brillo, histograma, etc.). Al trabajar con características predefinidas, la red solo se encarga de aprender la transformación de estas características en la salida esperada y se reduce el número de pesos a ajustar, ya que las características extraídas suelen tener menor dimensión que los datos originales.

Las redes neuronales obtenían mejores resultados utilizadas bajo la perspectiva del *machine learning* que del *deep learning*. Sin embargo, esto cambió cuando se co-



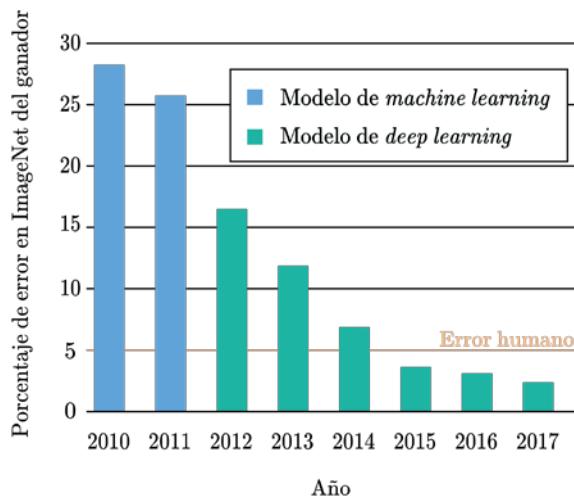
**Figura 4.4:** Esquema completo de un modelo de *deep learning* con sus diferentes etapas

menzaron a utilizar un nuevo tipo de red, conocida como **red neuronal convolucional**, que se estudia en detalle en la Sección 4.2. Las redes neuronales convolucionales sí son invariantes a la posición y pueden detectar características en cualquier posición de la entrada, sin necesitar por ello un mayor número de pesos.

El hito que puso al *deep learning* en el mapa fue la aplicación de la red neuronal convolucional, AlexNet [20], creada por Geoffrey Hinton y un alumno suyo de doctorado en el año 2012, en la competición de clasificación de imágenes ImageNet [21]. Esta competición exige la clasificación de imágenes entre más de veinte mil clases diferentes con más de diez millones de imágenes disponibles para entrenar. Las técnicas tradicionales de *machine learning* eran incapaces de bajar del 20 % de error, pero, AlexNet, fue capaz de bajarlo al 15.3 %, lo que captó automá-

ticamente el interés de toda la comunidad por este tipo de técnicas de *deep learning* debido a sus resultados.

Otra de las novedades que introdujo AlexNet fue el uso de las GPUs para el entrenamiento de las redes neuronales. Aprovechando el gran paralelismo de las GPUs, fue posible entrenar una red con millones de parámetros en un par de días, algo inaudito en aquel entonces. Desde que se popularizó el uso de las redes neuronales convolucionales en ImageNet, el error ha seguido bajando año tras año, llegando, en 2015, a bajar del considerado error humano (5 %) en la clasificación de imágenes [9]. En la Figura 4.5 se muestra la evolución del porcentaje de error cometido por el modelo ganador en ImageNet a lo largo de los últimos años.



**Figura 4.5:** Evolución del porcentaje de error cometido por del modelo ganador de ImageNet a lo largo de los últimos años

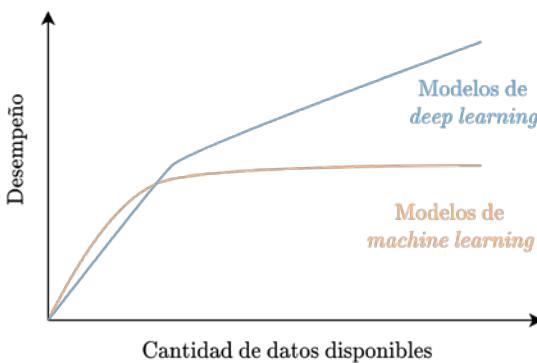
La principal ventaja de las técnicas de *deep learning* respecto a las de *machine learning* es que facilitan el desarrollo de los modelos. Ya no es necesario invertir una elevada cantidad de horas de trabajo de expertos en diseñar características. Además, los sistemas basados en *machine learning* cuentan con un sesgo previo que supone

una cota superior en el desempeño del sistema: el experto que elige las características se encuentra limitado en cuanto a la cantidad de información que es capaz de manejar y el hecho de que se añadan más datos puede ser contraproducente, ya que se hace más difícil discernir y encontrar las características más relevantes. Sin embargo, los sistemas de *deep learning* carecen de esta limitación y pueden cambiar las características si, al recibir más datos, lo encuentran oportuno para mejorar sus resultados. Al proveer al algoritmo con más datos es más fácil que el modelo generalice y responda mejor ante nuevos datos, algo que con las técnicas tradicionales no estaba garantizado y es donde radica su éxito.

No obstante, ante problemas más sencillos y donde se dispone de menos datos, las técnicas de *deep learning* tienen problemas para entrenarse correctamente. Tienen, comparado con modelos más sencillos, a sobreajustarse al conjunto de datos, además de ser mucho más costoso su entrenamiento. Esto explica en parte por qué, hasta prácticamente el 2012, no se ha utilizado de forma más general este tipo de técnicas para resolver problemas. El abaratamiento del almacenamiento, la mayor disponibilidad de datos gracias a Internet (ver Figura 4.7), el aumento de capacidad de cómputo y el uso de las GPUs para el entrenamiento de los modelos de *deep learning*, junto con los resultados que estos consiguen, han propiciado el auge de este campo. En la Figura 4.6 se muestra una comparativa del rendimiento de los modelos de *machine learning* y *deep learning* en base al tamaño del conjunto de datos disponible.

A pesar de pasar desapercibidos en su día, los investigadores que trabajaron en los comienzos del *deep learning* cuentan hoy con el reconocimiento que se merecen. Cabe destacar las figuras de Yann LeCun, director del Laboratorio de Inteligencia Artificial de Facebook y creador de la primera red neuronal convolucional de la historia; Yoshua Bengio, investigador de la Universidad de Montreal y director del Instituto de Algoritmos de Aprendizaje de Montreal (MILA); y Geoffrey Hinton, investigador de Google Brain y de la Universidad de Toronto, miembro de la Royal Society y considerado el padrino del *deep learning*. Todos ellos fueron galardonados con el Premio Turing (considerado como el premio Nobel de la informática) en 2018 [22] por sus avances en este campo. En la Figura 4.8 se encuentra una imagen de los ganadores.

El auge de esta tecnología ha traído consigo un nuevo tejido empresarial a su alrededor y ha potenciado el que ya existía basado en *machine learning*. Merecen una



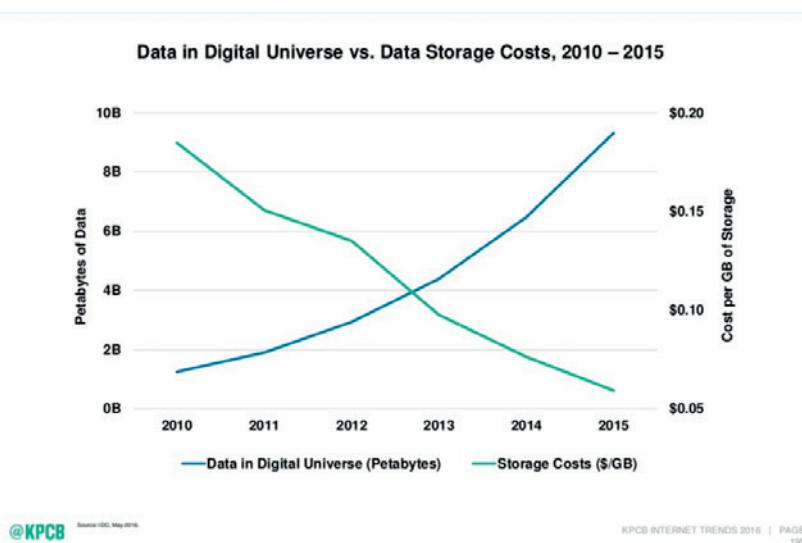
**Figura 4.6:** Comparativa del rendimiento de los modelos de *machine learning* y *deep learning* en base al tamaño del conjunto de datos disponible

mención especial empresas como Deep Mind, que se dedica a la investigación puntera en *deep learning*, o TensorFlow, que es uno de los *frameworks* más populares para trabajar en *deep learning*. En la Figura 4.9 se recoge el panorama industrial actual de este campo.

Este ecosistema empresarial ha propiciado la adopción de las técnicas de *deep learning* para aplicaciones tales como la automatización industrial, la lucha contra el cáncer y la conducción autónoma, además un largo etcétera.

Uno de los momentos más relevantes de la historia reciente del *deep learning* se produjo en 2016 cuando AlphaGo [23], un sistema basado en técnicas de *deep learning* desarrollado por Deep Mind, logró derrotar al campeón mundial de Go, Lee Sedol. Este juego tiene una complejidad de varios órdenes de magnitud superior a la de juegos como el ajedrez y se consideraba imposible que una inteligencia artificial fuera capaz de jugar correctamente a este juego. Este hito recuerda al ocurrido durante la década de los 90, en el que la inteligencia artificial de IBM, Deep Blue, logró vencer al campeón humano de ajedrez, Garry Kaspárov.

Si bien el *deep learning* ha traído consigo la democratización de las técnicas de inteligencia artificial, permitiendo, con un ordenador doméstico y datos recabados



**Figura 4.7:** Coste del almacenamiento de los datos frente a su disponibilidad

Fuente: <https://www.kleinerperkins.com/perspectives/2016-internet-trends-report>

por cualquier persona u organización, desarrollar un sistema de *deep learning* que mejore los resultados de técnicas tradicionales, aún sigue existiendo una gran diferencia entre lo que pueden conseguir pequeñas empresas o usuarios y grandes empresas como Google o Facebook. Estas últimas cuentan con miles de servidores para realizar multitud de pruebas con diferentes estrategias, que les permite encontrar mejores modelos para un problema, que es lo que realmente marca la diferencia y permite conseguir mejores resultados [24]. Además, la alta demanda de procesamiento de estos modelos están produciendo serios problemas de huella de carbono [25].



**Figura 4.8:** De izquierda a derecha: Yann LeCun, Geoffrey Hinton y Yoshua Bengio

Fuente: <https://www.forbes.com/sites/nicolemartin1/2019/03/27/turing-award-and-1-million-given-to-3-ai-pioneers>

## Deep Learning – Fundamentos, teoría y aplicación



**Figura 4.9:** Panorama industrial del deep learning

Fuente: <http://www.shivonzilis.com/machineintelligence>

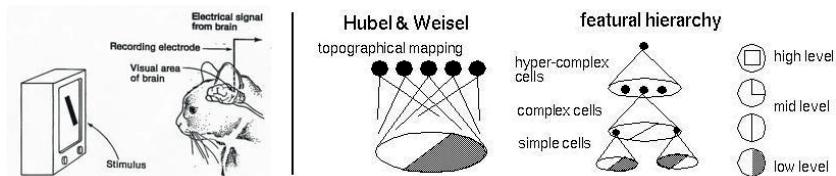
## 4.2. Redes neuronales convolucionales

Las redes neuronales convolucionales, *convolutional neural networks* (CNN) en inglés, son un modelo de red neuronal inspirado en el neocórtez (la parte del cerebro encargada de las funciones más complejas que realizan los humanos), en concreto, en la forma en la que la corteza visual procesa la información.

En el año 1959, los profesores David Hubel y Torsten Wiesel desarrollaron un trabajo experimental en el que registraron la activación de las neuronas de la corteza visual del cerebro de un gato mientras este veía una serie de dibujos [26]. Para su sorpresa, encontraron que había neuronas que se activaban si la línea que dibujaban tenía una orientación determinada, a las que llamaron células simples; mientras que otras se activaban si, además de presentar una orientación determinada, se desplazaban con una orientación específica. A estas últimas las llamaron células complejas. Observaron que el cerebro analiza la información visual mediante el uso de estas células, organizadas en una estructura jerárquica donde las células que están próximas analizan una región cercana del campo visual.

Las células simples reaccionan a las características más básicas como bordes o líneas en una determinada orientación, mientras que las células complejas se basan en las características detectadas por las células simples para detectar características más complejas. En concreto, descubrieron que las células complejas ofrecen una cierta invarianza a la localización de las características gracias a la agrupación que realizan de las entradas recibidas de varias células simples. En la Figura 4.10 se muestra un resumen de los experimentos llevados a cabo por Hubel y Wiesel.

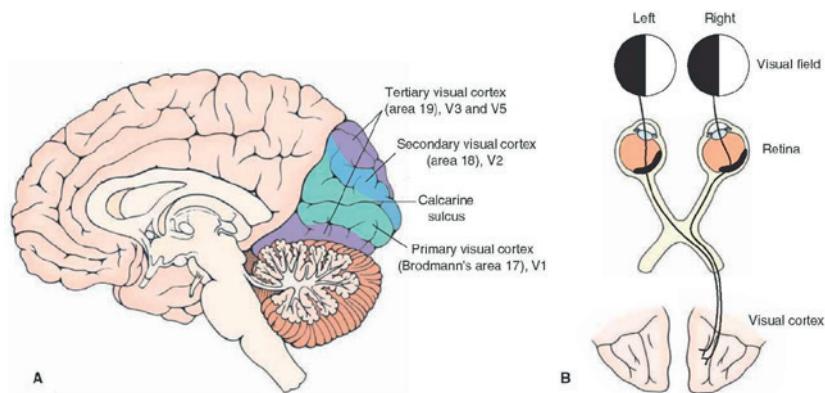
En los humanos, la corteza visual se localiza en la parte posterior del cerebro (ver Figura 4.11). En ella se distinguen diferentes capas encargadas de procesar la información visual. Cada capa extrae características basadas en las extraídas por las capas anteriores, a la vez que va aumentando el campo receptivo, es decir, la región del espacio que es procesada por cada neurona. Esta información es luego enviada a otras regiones del cerebro para que siga el procesamiento de la información a más alto nivel y sea de ayuda en la toma de decisiones. En la Figura 4.11 se muestran las



**Figura 4.10:** Experimentos llevados a cabo por Hubel y Wiesel

Fuente: <https://ml4a.github.io/ml4a/convnets/>

principales características que se extraen en cada región de la corteza visual durante el procesamiento de la información que recibe de la retina.



**Figura 4.11:** Localización de la corteza visual en el cerebro

Fuente: <https://www.pinterest.es/pin/720998221568035725/>

El trabajo de Hubel y Wiesel, donde describían cómo el cerebro de los mamíferos procesa la información visual mediante la extracción de características de menor a mayor complejidad, les valió para obtener el premio Nobel, además de sentar las bases para los modelos de *deep learning*.

Uno de los primeros modelos que se desarrollaron a partir de este trabajo fue el **Neocognitron**, que sirvió de inspiración para las redes neuronales convolucionales.

les que se conocen hoy día. Kunihiko Fukushima presentó en 1980 un modelo de red neuronal llamado **Neocognitron** [27]. Este modelo, basado en el trabajo de Hubel y Wiesel, es capaz de procesar una imagen mediante una serie de capas, encargadas de extraer características, dispuestas de forma secuencial con el objetivo de imitar la extracción de características jerárquica del cerebro. Para ello, cada capa del Neocognitron emplea dos tipos de células: las células S y las células C. Las células S extraen características de bajo nivel de la entrada, como las células simples del cerebro; y las células C extraen características de alto nivel, como las células complejas del cerebro. En una capa, las células se organizan en unos planos donde se distribuyen en una cuadrícula de forma que cada una genera un valor de salida que permite mantener la estructura de los datos en forma matricial (como la imagen de entrada). Las células que pertenecen a un mismo plano extraen la misma característica de la entrada (todas comparten los mismos pesos) solo que cada una analiza una región diferente.

Al contrario que en las redes neuronales tradicionales, se respeta la estructura espacial de los datos (la imagen en este caso) y no se convierte la entrada en un vector. Cada célula procesa una pequeña matriz (una pequeña región de la imagen de entrada) mediante una operación de **convolución** (ver Sección 4.2.1.1), que le permite comparar la región actual con la plantilla (pesos) de la característica que detecta la célula (ver Figura 4.13). Las células C utilizan una operación de agrupamiento o *pooling* en inglés (normalmente la media) para combinar la salida de diferentes células S y así ganar algo de invarianza ante pequeñas traslaciones o rotaciones de las características detectadas por las células S. En la Figura 4.14 se muestra la estructura de este modelo.

Las principales aportaciones del Neocognitron fueron el procesado de la entrada mediante la operación de convolución para detectar características, preservar la estructura espacial de los datos y la tolerancia a traslaciones de las características. Esto último se consigue gracias al uso de operaciones como el *pooling* y a utilizar los mismos pesos en diferentes posiciones de la entrada, lo que reduce la cantidad total de pesos que hay que ajustar. Estos principios son en los que se basan las **redes neuronales convolucionales**.

En el año 1989, Yann LeCun y su equipo del laboratorio AT&T Bell, presentaron la primera red neuronal convolucional de la historia, llamada LeNet-5, que fue en-

trenada mediante el algoritmo *backpropagation* (que no existía cuando se presentó el Neocognitron) para el reconocimiento de dígitos escritos a mano [28].

Esta nueva red neuronal, a diferencia de la red neuronal tradicional que solo emplea capas con neuronas, cuenta con otros tipos de capas, que se utilizan antes de las capas de neuronas para extraer características de los datos en bruto de entrada. Estas primeras capas, normalmente de convolución y *pooling*, realizan una transformación del espacio de los datos de entrada a un nuevo espacio de características de una dimensión mucho más reducida, que es utilizada por una red neuronal tradicional para dar la salida de la red, en lugar de utilizar directamente los datos de entrada.

Por lo tanto, se distinguen dos etapas principales en una red neuronal convolucional: una etapa de extracción de características y una etapa de clasificación. En la etapa de extracción de características las capas de convolución y *pooling* (ver Sección 4.2.1.2) obtienen una nueva representación de los datos de entrada basada en la creación de una jerarquía de características de menor a mayor complejidad. La salida de la última capa de esta etapa es la nueva representación obtenida de los datos de entrada y es la entrada de la siguiente etapa de clasificación. La clasificación no es más que una red neuronal tradicional, solo que, en el caso de las redes neuronales convolucionales, su entrada no es directamente los datos del problema sino la salida de la etapa anterior.

La ventaja de este enfoque es que la entrada de la etapa de clasificación es mucho más reducida que en el caso tradicional, lo que disminuye el número de parámetros que tiene que ajustar la red. Además, la nueva representación de los datos facilita la clasificación final de la red gracias a que cada convolución detecta características diferentes, lo que permite representar a los datos de entrada como la combinación de estas. La presencia de ciertas características en un dato de entrada es lo que combina la red neuronal para determinar la clase de la entrada.

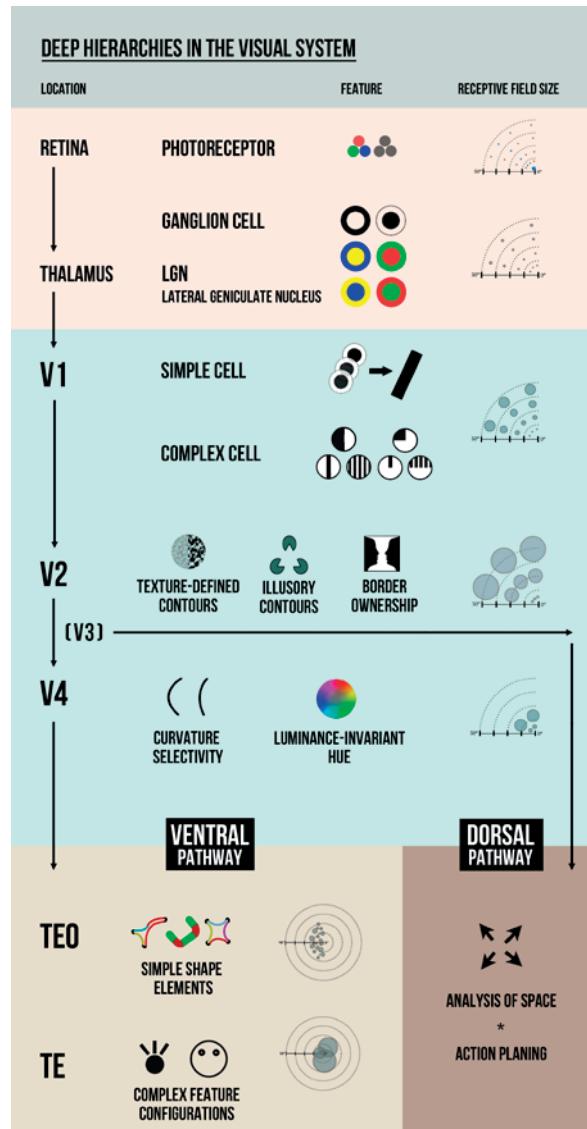
En ambas etapas, el algoritmo *backpropagation* se encarga de ajustar los pesos: las características que extraen las primeras capas se definen de forma que las capas de neuronas finales de la etapa de clasificación puedan generar la salida correctamente para las entradas.

En base a estudios recientes, se ha observado de forma empírica [29] que las primeras capas de una red neuronal convolucional extraen características de bajo nivel

(bordes, colores, etc.) que sirven de entrada para las siguientes capas donde se utilizan para detectar características más complejas (como coches o caras), lo que guarda similitud con lo observado en las primeras capas de la corteza visual de nuestro cerebro [30]. Al ir avanzando en las capas de la red, cada operación de convolución va extrayendo características más complejas, que se traducen en una mejor representación distribuida de los datos de entrada para su clasificación. En las Figuras 4.15, 4.16 y 4.17 se muestran las características que detectan diferentes capas de convolución de una red neuronal convolucional. Se observa cómo las capas más profundas de la red detectan características más complejas (ruedas de un coche o personas), mientras que la primera y segunda capa detectan características más simples (color y bordes en la primera y esquinas y círculos en la segunda).

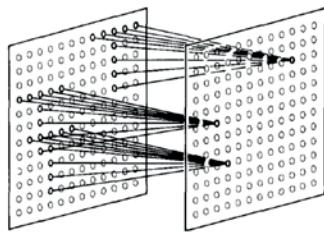
A pesar de que todos los pesos de la red neuronal convolucional se ajustan de forma automática mediante *backpropagation* y de conseguir excelentes resultados en el problema de reconocimiento de dígitos, la mayoría de la comunidad pensó que no sería posible alcanzar buenos resultados en tareas más complejas, con más clases y mayor variabilidad de las imágenes. No fue hasta el año 2012, cuando una nueva red neuronal convolucional, AlexNet, se utilizó en la competición ImageNet, reduciendo considerablemente el error del mejor modelo del año anterior, lo que provocó la popularización de las redes neuronales convolucionales y, en definitiva, de las técnicas de *deep learning*.

Desde entonces, las redes neuronales convolucionales han sido aplicada a multitud de problemas y aplicaciones. Modificando su arquitectura, una red neuronal convolucional puede ser aplicada, además de a la clasificación de imágenes, a la segmentación o detección de objetos en imágenes. Por otro lado, la transformación de los datos a una representación con estructura espacial (como las imágenes) ha permitido el procesamiento del audio o del texto por parte de las redes neuronales convolucionales. Esto ha mejorado los resultados conseguidos por las técnicas tradicionales en estos campos de aplicación, gracias a no tener que definir características a mano y ser la propia red la que las define.



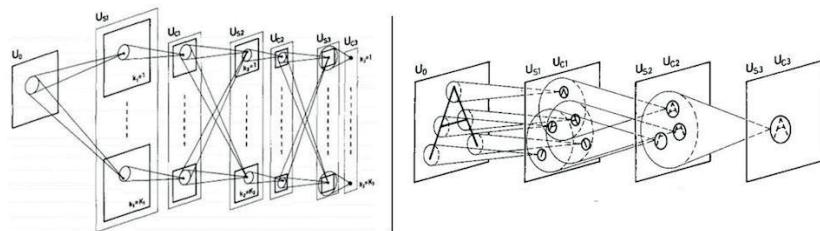
**Figura 4.12:** Jerarquía de características que obtiene el cerebro a partir de la información visual de la retina

Fuente: [https://en.wikibooks.org/wiki/Sensory\\_Systems/Visual\\_Signal\\_Processing](https://en.wikibooks.org/wiki/Sensory_Systems/Visual_Signal_Processing)



**Figura 4.13:** Células S que comparten pesos para extraer la misma característica en diferentes regiones de la entrada

Fuente: <http://www.rctn.org/bruno/public/papers/Fukushima1980.pdf>



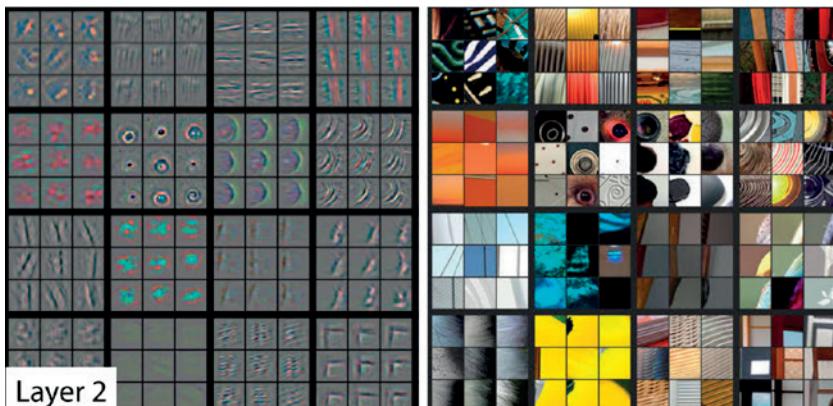
**Figura 4.14:** Estructura del Neocognitron

Fuente: <https://ml4a.github.io/ml4a/convnets/>



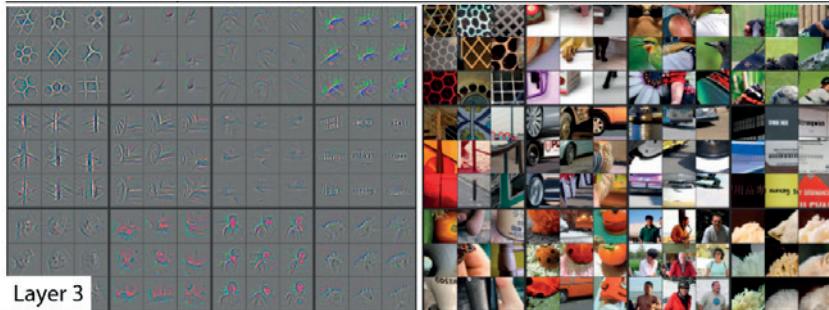
**Figura 4.15:** Características extraídas por la primera capa de una red neuronal convolucional. Arriba se muestra la imagen que produce mayor activación de la característica y, debajo, ejemplos de imágenes reales del conjunto de datos que producen una alta activación.

Fuente: <https://arxiv.org/pdf/1311.2901.pdf>



**Figura 4.16:** Características extraídas por la segunda capa de una red neuronal convolucional. A la izquierda se muestra la imagen que produce mayor activación de la característica y, a la derecha, ejemplos de imágenes reales del conjunto de datos que producen una alta activación.

Fuente: <https://arxiv.org/pdf/1311.2901.pdf>



**Figura 4.17:** Características extraídas por la tercera capa de una red neuronal convolucional. A la izquierda se muestra la imagen que produce mayor activación de la característica y, a la derecha, ejemplos de imágenes reales del conjunto de datos que producen una alta activación.

Fuente: <https://arxiv.org/pdf/1311.2901.pdf>

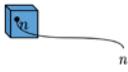
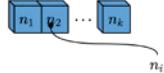
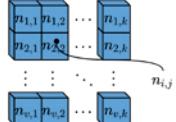
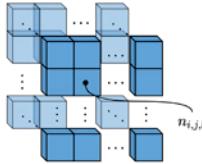
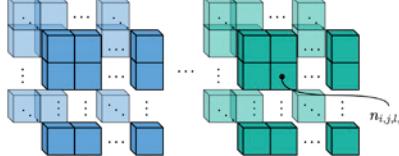
### 4.2.1. Arquitectura

La arquitectura de una red neuronal convolucional difiere de la arquitectura de las redes neuronales tradicionales. Además de contar con diferentes tipos de capa, la estructura de los datos pasa a tener un papel relevante en el procesamiento que realiza la red. Siempre que es posible, la estructura espacial de los datos se preserva, tanto para facilitar la extracción de características como para generar la salida de la red con la estructura esperada. En el primer caso, las capas de convolución o *pooling* aprovechan la relación espacial que presentan los datos para extraer características. En el segundo caso, dependiendo del objetivo de la red, la estructura de los datos se puede modificar para que cuenten con la representación deseada. Esto presenta una novedad respecto de las redes neuronales en las que tanto la entrada como la salida es siempre un vector.

Cuando se trabaja con las redes neuronales tradicionales, cada neurona recibe como entrada cada uno de los elementos del vector de entrada. Este vector de entrada tiene una dimensión (1D). Si hay un único dato de entrada, este tiene cero dimensiones (0D), es decir, es un escalar. Cuando los datos tienen dos dimensiones (2D), como en el caso de una imagen en escala de grises, los datos tienen una representación matricial. Para ser procesados por la red neuronal, independientemente de la representación original de los datos, estos se transforman en un vector. Para el caso de datos con tres dimensiones (3D), como imágenes a color, o de más dimensiones, se sigue el mismo procedimiento de transformar los datos en un vector de una dimensión. Esto no ocurre en el caso de las redes neuronales convolucionales, estas pueden respetar las dimensiones de los datos o modificarlas, por lo que es importante conocer siempre la estructura que tienen los datos en todo momento para aplicar las operaciones correctamente.

De forma general, y para no tener que distinguir entre escalar, vector o matriz; se utiliza el concepto de tensor para hacer referencia a los datos. De esta forma, un escalar es un tensor de orden cero, un vector un tensor de orden uno, una matriz un tensor de orden dos y, de forma general, un dato de dimensión  $n$  es un tensor de orden  $n$ . En la Figura 4.18 se muestran ejemplos de diferentes datos, su representación y su denominación en base a su dimensión.

El diseño de la arquitectura de la red debe tener en cuenta el tipo de operación y el tipo de tensor que se procesa para programarla correctamente.

Dato	Representación	Dimensión-Tensor
Temperatura, velocidad, edad, saldo de cuenta bancaria, ...		Escalar Dimensión 0 $\equiv$ Tensor de orden 0
Temperatura diaria, velocidad en un trayecto, edad de cada cliente, ...		Vector Dimensión 1 $\equiv$ Tensor de orden 1
Imagen en escala de grises		Matriz Dimensión 2 $\equiv$ Tensor de orden 2
Imagen a color (RGB), Imagen hiperespectral, ...		Dimensión 3 $\equiv$ Tensor de orden 3
Video		Dimensión 4 $\equiv$ Tensor de orden 4

**Figura 4.18:** Ejemplos de diferentes datos, su representación y la denominación que tienen en base a su dimensión

Normalmente, para extraer características se utilizan sucesiones de capas de convolución y *pooling*, mientras que para la etapa de transformación se utilizan capas de redes neuronales tradicionales. Decidir el tipo y combinación de capas en una red neuronal convolucional requiere de un proceso de prueba y error hasta encontrar la mejor combinación. Para esta tarea es conveniente revisar la literatura en busca de soluciones contrastadas para problemas parecidos o con buenos resultados en problemas complejos. Analizando las arquitecturas más populares a lo largo de la

historia se podrá entender el porqué de muchos de los estándares que actualmente se siguen al diseñar la arquitectura de una red neuronal convolucional. En cualquier caso, siempre se puede diseñar una arquitectura desde cero o, cuando se encuentre una arquitectura de red que se considere adecuada para el problema en la literatura, replicar su arquitectura o intentar modificarla para mejorar sus resultados.

A pesar de que se pueden combinar los diferentes tipos de capas como se desee, se observa, al estudiar algunas arquitecturas populares, que suelen diseñarse, en algunos aspectos, de forma parecida. Por ejemplo, cuando se hace uso de redes neuronales, siempre que es posible, se intenta reducir la dimensión de los datos para evitar que aumenten de forma excesiva el número de parámetros de la red. Esta y otras *buenas prácticas* se estudiarán en la Sección 4.2.3.

En las siguientes secciones se estudian las principales capas de una red neuronal convolucional.

#### 4.2.1.1. Capa de convolución

La capa de convolución es una de las capas más importantes de una red neuronal convolucional. Esta capa permite extraer características de los datos de entrada. Para ello, la capa de convolución aplica una serie de convoluciones a los datos de entrada, pudiendo ser la entrada de la red o la salida de alguna otra capa. Dado que la capa de convolución busca extraer características, se hace referencia a la salida y la entrada como mapa de características de entrada o de salida según corresponda.

**La operación de convolución.** La convolución es un operador matemático que se representa con el símbolo  $*$ . Este operador realiza la transformación de dos funciones,  $f$  y  $g$ , en una tercera función,  $(f * g)$ , mediante la integración del producto de las dos funciones con una de ellas invertida y desplazada cierta distancia  $x$ :

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

A la función que es invertida y desplazada se la denomina *kernel* de convolución.

La intuición detrás de la convolución es que permite obtener una medida de la superposición de ambas funciones. En la Figura 4.19 se muestra de forma visual la convolución de dos funciones de ejemplo. Como se observa en la figura, la convolución da el área bajo la curva del producto de las dos funciones en cada punto.

La definición anterior de convolución es la utilizada para funciones continuas. Sin embargo, los datos que son procesados por una red neuronal o una red neuronal convolucional son discretos: histórico del precio en bolsa de un activo, texto de opinión en una red social, imágenes, etc. Para estos casos, se hace uso de la operación de convolución discreta. Esta operación sustituye la integración por el sumatorio y se calcula como

$$(f * g)(x) = \sum_i f(i) * g(x - i)$$

En la Figura 4.20 se muestra la convolución de dos funciones discretas de ejemplo. A diferencia de la convolución de funciones continuas, en la función discreta solo se dispone de valores en los puntos de muestreo de la función.

En el ejemplo anterior, ambas funciones tienen un dominio infinito, pero este no será el caso cuando se trabaje con redes neuronales convolucionales: tanto el mapa de características de entrada como el *kernel* de convolución tienen un tamaño finito. A estos dos elementos también se pueden hacer referencia como piezas de información en lugar de funciones cuando se habla de la convolución de ambos en una red neuronal convolucional.

La representación de la convolución discreta de dos piezas de información con un tamaño finito no se suele representar como se ha visto hasta ahora. En su lugar, se representan de forma similar a como se representan las imágenes, es decir, utilizando una disposición en cuadrícula donde se sitúa cada elemento del mapa de características o del *kernel* de convolución con la posición que tengan, respetando la estructura de los datos. En la Figura 4.21 se muestra un ejemplo de esta representación para la convolución de dos piezas de información de una dimensión, con un tamaño limitado y diferente.

A pesar de que la convolución es conmutativa, se suele utilizar como *kernel* de convolución a la pieza de información más pequeña.

A diferencia de las situaciones anteriores, cuando la convolución se aplica sobre piezas de información con un tamaño finito puede ocurrir que el resultado de la convolución no tenga el mismo tamaño que las funciones de entrada. Si se observa la Figura 4.21, el resultado de la convolución tiene 8 elementos, pero las dos funciones sobre las que se aplica la convolución cuentan con 4 y 11 elementos respectivamente. Esto es debido a que la convolución solo se puede calcular en las posiciones donde ambas funciones están definidas, es decir, cuando el *kernel* de convolución se sitúa completamente sobre la otra función. Esto provoca que el resultado de la convolución tenga un tamaño más reducido que el de la función sobre la que se aplica la operación.

Cuando es necesario mantener el tamaño original de la pieza de información sobre la que se desplaza el *kernel* de convolución, se suele introducir un **relleno**, *padding* en inglés, a la pieza de información para que el resultado de la convolución tenga el tamaño deseado. Normalmente, el relleno consiste en introducir ceros por ambos extremos de la pieza de información. En la Figura 4.22 se muestra un ejemplo de convolución discreta sobre dos piezas de información con tamaño finito donde se ha aplicado un relleno de ceros para mantener el tamaño original la pieza de información a la salida de la convolución.

Además de poder modificar el tamaño de la función sobre la que se aplica la convolución para que sea más grande mediante la adición de relleno, también se puede reducir el tamaño. Para ello, se introduce el concepto de **paso**, *stride* en inglés. Por norma general, la convolución se aplica sobre todas las posiciones de la función, es decir, el paso que da la convolución es de 1. Sin embargo, si se quiere reducir el tamaño de la salida, se puede elegir un paso mayor y el tamaño de la salida se reducirá en proporción al paso elegido. Si el mapa de características tiene tamaño  $n$ , el *kernel* de convolución tiene tamaño  $k$  y el paso de la convolución es  $p$ , el tamaño de salida será

$$\left\lfloor \frac{n - k}{p} \right\rfloor + 1$$

En la Figura 4.23 se muestra un ejemplo de convolución discreta sobre dos piezas

de información de tamaño finito donde se ha utilizado un paso de 2. En este caso, el resultado de la convolución tiene tamaño 4, ya que

$$\left\lfloor \frac{n - k}{p} \right\rfloor + 1 = \left\lfloor \frac{11 - 4}{2} \right\rfloor + 1 = 4$$

Hasta ahora, se ha estudiado la convolución aplicada sobre funciones de una dimensión, pero se podría aplicar a cualquier número de dimensiones. Por ejemplo, la convolución discreta en dos dimensiones se calcula como

$$(f * g)(x, y) = \sum_i \sum_j f(i, j) * g(x - i, y - j)$$

Un ejemplo familiar de la convolución en dos dimensiones es el filtrado de imágenes, en este caso, al tratarse de dos dimensiones, imágenes en escala de grises. En la Figura 4.24 se muestra un ejemplo de convolución en dos dimensiones. Al representar el mapa de características como una imagen en escala de grises (mediante su normalización) se puede apreciar el efecto que tiene el filtro elegido sobre la entrada. En el caso de la Figura 4.24, el *kernel* de convolución es capaz de resaltar los ojos de la figura dibujada. Si se observa el mapa de características resultado de aplicar la convolución se aprecia que el valor más alto se da en las posiciones donde se encuentran los ojos de la figura.

Para datos de más de dos dimensiones, como es el caso de las imágenes a color (tres dimensiones), el efecto de la convolución sobre las imágenes es más fácil de apreciar. En la Figura 4.25 se muestran algunos ejemplos del resultado de convolucionar con diferentes *kernels*, muchos de ellos ya conocidos en el campo del procesamiento de imágenes, una imagen a color. Tal y como se observa en la figura, según el *kernel* elegido se puede desde desenfocar la imagen a detectar bordes en la dirección cualquier dirección.

Una de las principales apreciaciones en el ejemplo de la Figura 4.25 es que el resultado de la convolución es una imagen en escala de grises (dos dimensiones) a pesar de que la entrada y el *kernel* de convolución tienen tres dimensiones. Esto es debido a la definición de la propia operación de convolución, que calcula en cada posición

la suma total de todos los elementos del *kernel* con la porción correspondiente de la imagen de entrada, resultando en la pérdida de la última dimensión.

Como se ha observado en los ejemplos anteriores, según el *kernel* que se elija, la convolución extraerá determinadas características de los datos de entrada. El objetivo será obtener una definición de este *kernel* de modo similar a cómo se obtienen los pesos de la red neuronal tradicional, es decir, utilizando algún algoritmo de optimización.

En el campo del *deep learning*, aunque se la denomina convolución, la operación empleada en las redes neuronales convolucionales es la correlación cruzada, *cross-correlation* en inglés. Esta operación se calcula de forma similar a la convolución solo que no se refleja el *kernel* de convolución. Es decir, la correlación cruzada de dos funciones discretas,  $f$  y  $g$ , se calcula como

$$(f * g)(x) = \sum_i f(i) * g(x + i)$$

Dado que será la red neuronal convolucional la encargada de aprender los valores de los *kernels* de la convolución, es indiferente que aprenda el *kernel* antes de ser reflejado, como indica la operación de convolución, o después de ser reflejado, como en el caso de la correlación cruzada. Si se considera que la red va a aprender directamente el *kernel* reflejado, se puede multiplicar directamente por la entrada y se evita tener que reflejarlo. Es por esto que se dice que, realmente, la operación que se realiza en una capa de convolución es la correlación cruzada en lugar de la convolución.

**La capa de convolución.** Una capa de convolución se compone de una o más convoluciones. Al igual que en el caso de las neuronas de una red neuronal, las convoluciones que procesan la misma entrada se agrupan en una misma capa. La salida de una capa de convolución consiste en la concatenación de la salida de cada una de las convoluciones de la capa. Tomando el ejemplo de la Figura 4.25, si se considera que la capa de convolución la componen las cuatro convoluciones que se le aplica a la imagen de entrada, la salida de esta capa sería un mapa de características compuesto por la concatenación de cada una de las imágenes en escala de grises que han generado las convoluciones de la capa, formando una única imagen, es decir, una imagen de cuatro canales.

En la Figura 4.26 se muestra toda la información relativa a una capa de convolución. Es importante notar que el valor de relleno,  $r$ , se refiere al relleno total que se le añade al mapa de características de entrada. Sin embargo, este relleno suele dividirse y la mitad se aplica al comienzo y la otra mitad al final de cada dimensión donde se ha añadido relleno. Este relleno se aplica, normalmente, a las dimensiones donde se calcula la convolución ( $h$  y  $w$ ).

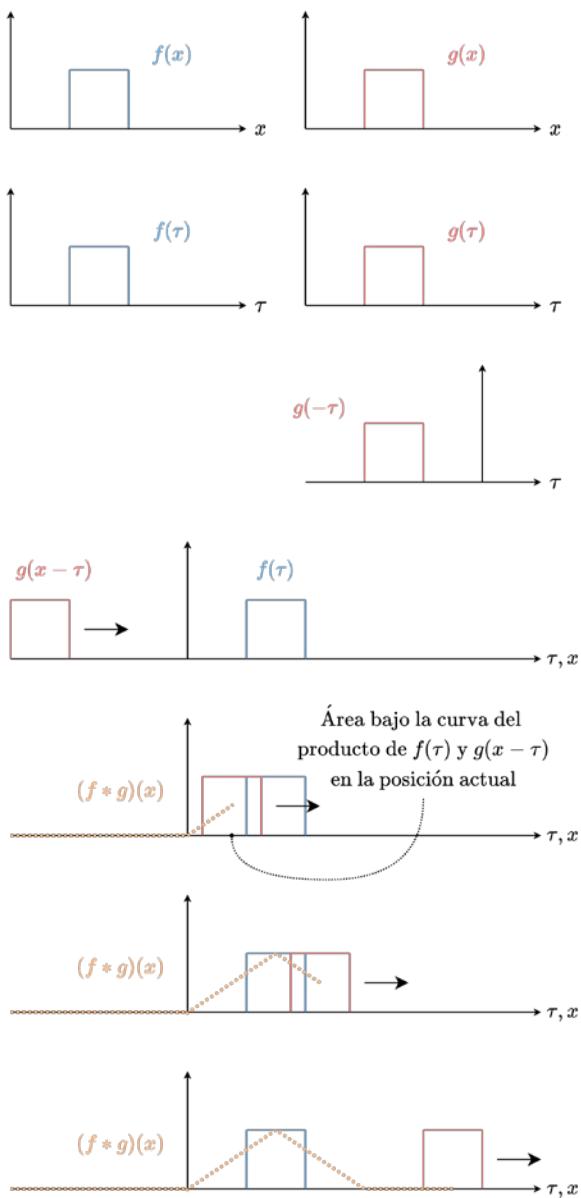
Los principales hiperparámetros de una capa de convolución son: el número de convoluciones, el tamaño del *kernel*, el tamaño del paso y la cantidad de relleno a emplear.

La ventaja del uso de la convolución respecto de una red neuronal es la invarianza a la traslación. Una misma característica (un borde p. ej.) puede ser detectado en cualquier posición de la entrada sin tener que emplear más parámetros que los específicos del *kernel* de convolución. La convolución desplaza el *kernel* por la entrada y calcula en cada posición la salida. Esto permite a la convolución utilizar el mismo *kernel* en diferentes posiciones. En una red neuronal, por el contrario, cada posición de la entrada se multiplica por un peso, lo que significa que una característica en la esquina superior izquierda de la entrada no es procesada por los mismos pesos que en la esquina inferior derecha. Para detectar la misma característica en ambas posiciones con una red neuronal habría que repetir los valores de los pesos en ambas posiciones. Esto supone otra de las ventajas de las redes neuronales convolucionales respecto de las redes neuronales tradicionales: el número de parámetros. Para una entrada de tamaño  $h \times w \times c$ , una capa de una red neuronal necesita  $h * w * c$  parámetros por cada una de las neuronas de la capa. Sin embargo, una capa de convolución con un *kernel* de tamaño  $k_h \times k_w \times k_c$  solo necesita  $k_h \times k_w \times k_c$  por cada convolución, independientemente del tamaño de la entrada. En la Figura 4.27 se muestra una comparativa entre una capa de una red neuronal y una capa de red neuronal convolucional.

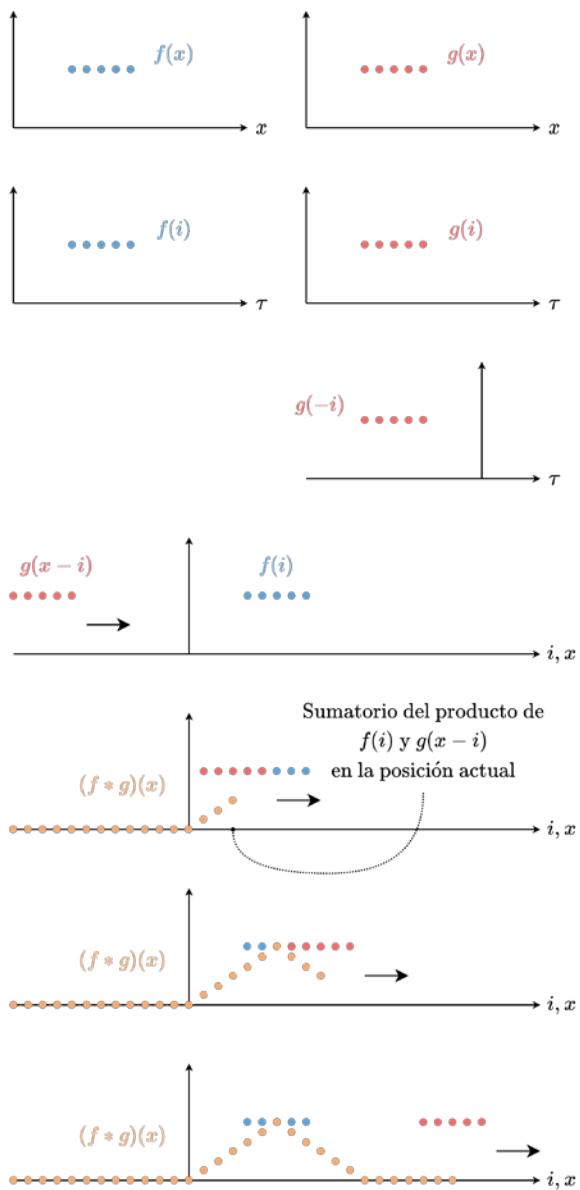
Otro aspecto importante a la hora de utilizar las capas de convolución es el uso de funciones de activación. Una convolución puede ser vista como una neurona si el tamaño del *kernel* coincide con el tamaño de la entrada. En este caso, cada elemento del *kernel* multiplica a uno de los datos de entrada, como si de una neurona tradicional se tratase. Por este motivo, al igual que en las redes neuronales se necesitan funciones de activación no lineales para mejorar la capacidad de la red, en las capas de convolución serán igual de necesarias. La función de activación se

aplica igual que en las redes neuronales tradicionales: una vez se obtiene el mapa de características de salida, cada uno de sus elementos son procesados por la función de activación y su valor actualizado, manteniendo la estructura de los datos. Las redes neuronales convolucionales utilizan las mismas funciones de activación que se suelen utilizar en las redes neuronales tradicionales, que ya se estudiaron en la Sección 1.2.1.

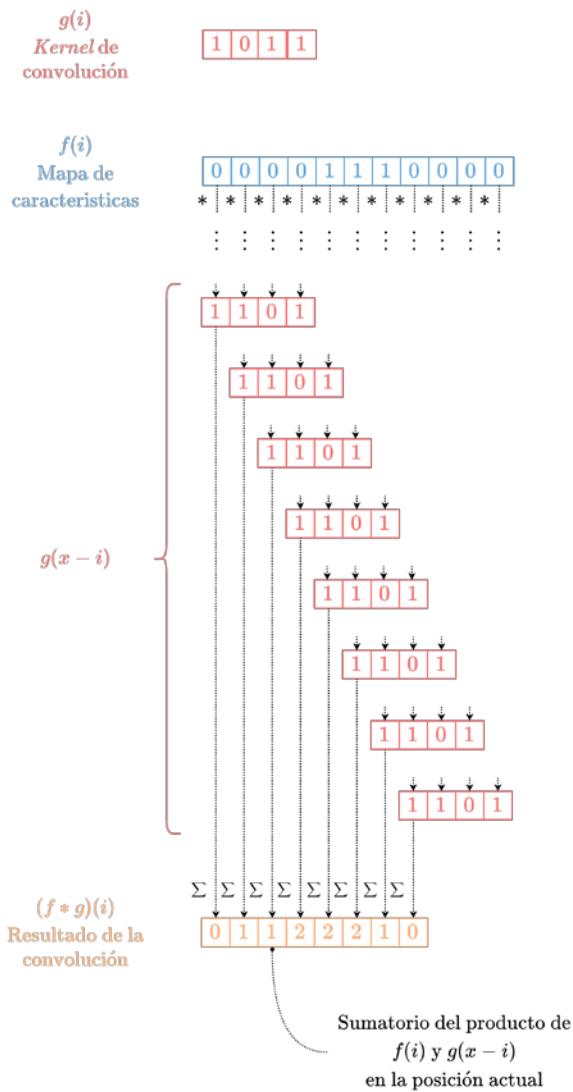
A la hora de diseñar la arquitectura de la red neuronal convolucional se tendrá que optar porque la red crezca en profundidad (añadiendo más capas) o en anchura (añadiendo más convoluciones a una capa). Como ya se ha comentado, la opción habitual suele ser aumentar la profundidad de la red. Este enfoque viene motivado por dos razones principales. La primera es la demostración práctica de la mejora de los resultados de las redes neuronales convolucionales al aumentar su profundidad en problemas genéricos, como la competición ImageNet (ver Figura 4.28). La segunda es el número de parámetros de la red. Si se utilizan pocas capas, se tendrá que utilizar un tamaño de *kernel* lo suficientemente grande para procesar toda la entrada. Por ejemplo, si se desea reconocer una persona en una imagen y esta ocupa toda la imagen, el tamaño del *kernel* debe ser, prácticamente, igual de grande que la imagen, con el incremento en el número de parámetros que conlleva. Por otro lado, si se utilizan varias capas de convolución con un *kernel* más pequeño, al cabo de varias capas, el *kernel* puede estar recibiendo como entrada información proveniente de la imagen completa. La zona de la entrada que una capa de convolución procesa se conoce como campo receptivo, *receptive field* en inglés, de la capa, y concuerda con el tamaño del *kernel*. Interesa mantener un compromiso entre el número de capas de la red y el campo receptivo, de forma que se controle el número de parámetros a ajustar por la red y se llegue a procesar el elemento de interés de la entrada.



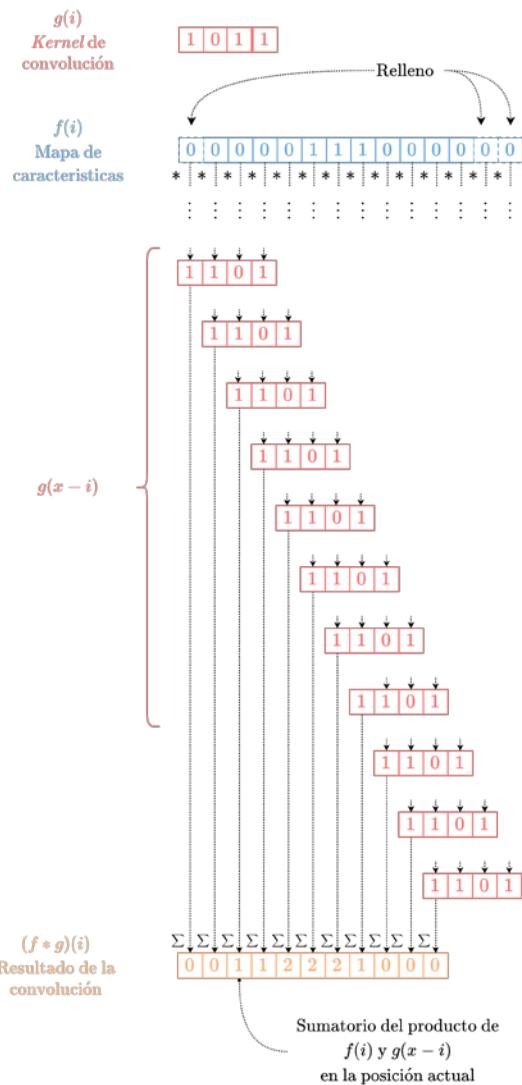
**Figura 4.19:** Ejemplo gráfico del cálculo de la convolución de dos funciones continuas



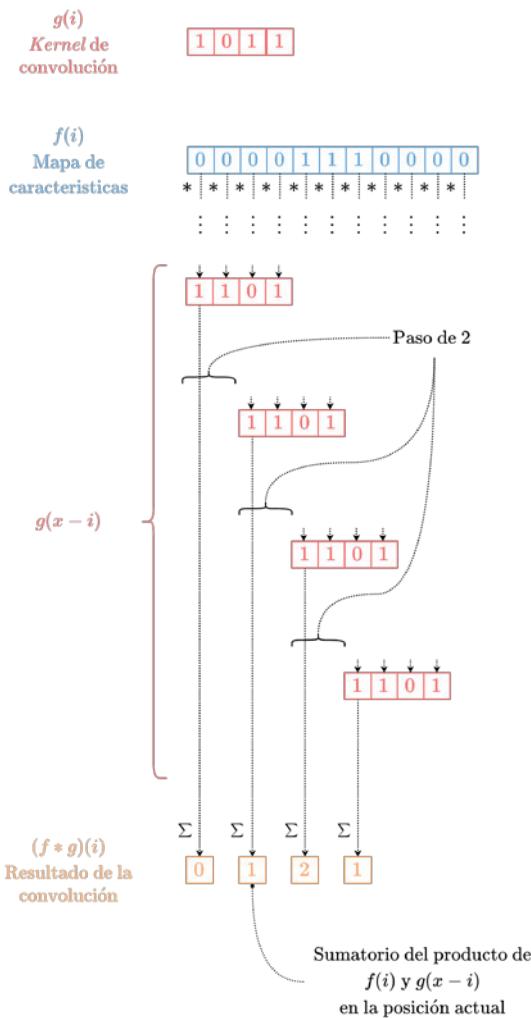
**Figura 4.20:** Ejemplo gráfico del cálculo de la convolución de dos funciones discretas



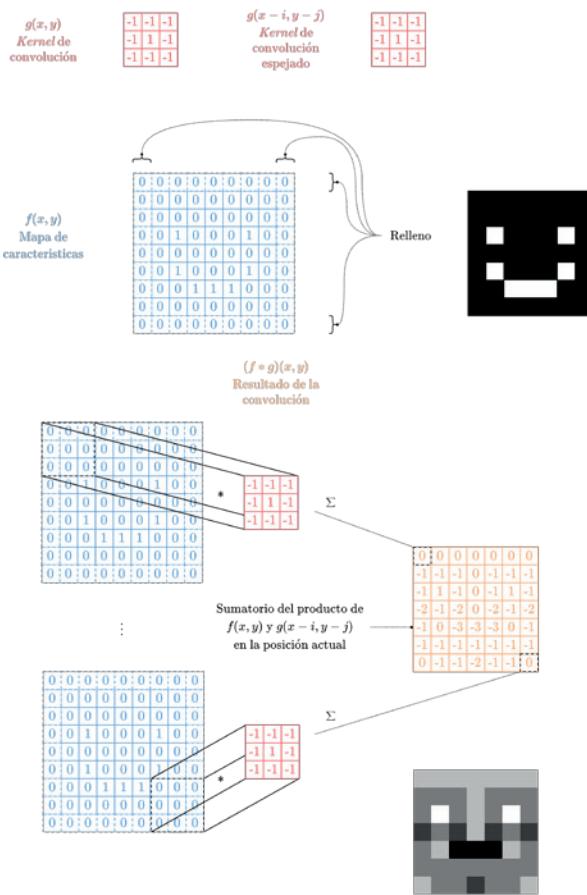
**Figura 4.21:** Representación de la convolución de dos piezas de información de una dimensión con tamaño limitado y diferente



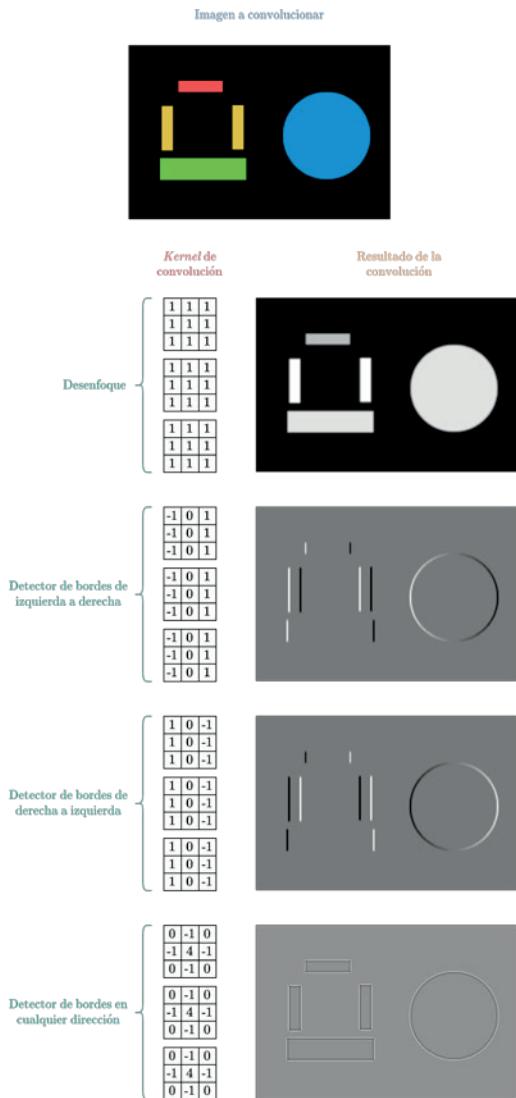
**Figura 4.22:** Representación de la convolución de dos piezas de información con relleno



**Figura 4.23:** Representación de la convolución de dos piezas de información con un paso de 2



**Figura 4.24:** Representación de la convolución de dos piezas de información de dos dimensiones



**Figura 4.25:** Diferentes convoluciones sobre una imagen de tres dimensiones (canales rojo, verde y azul)

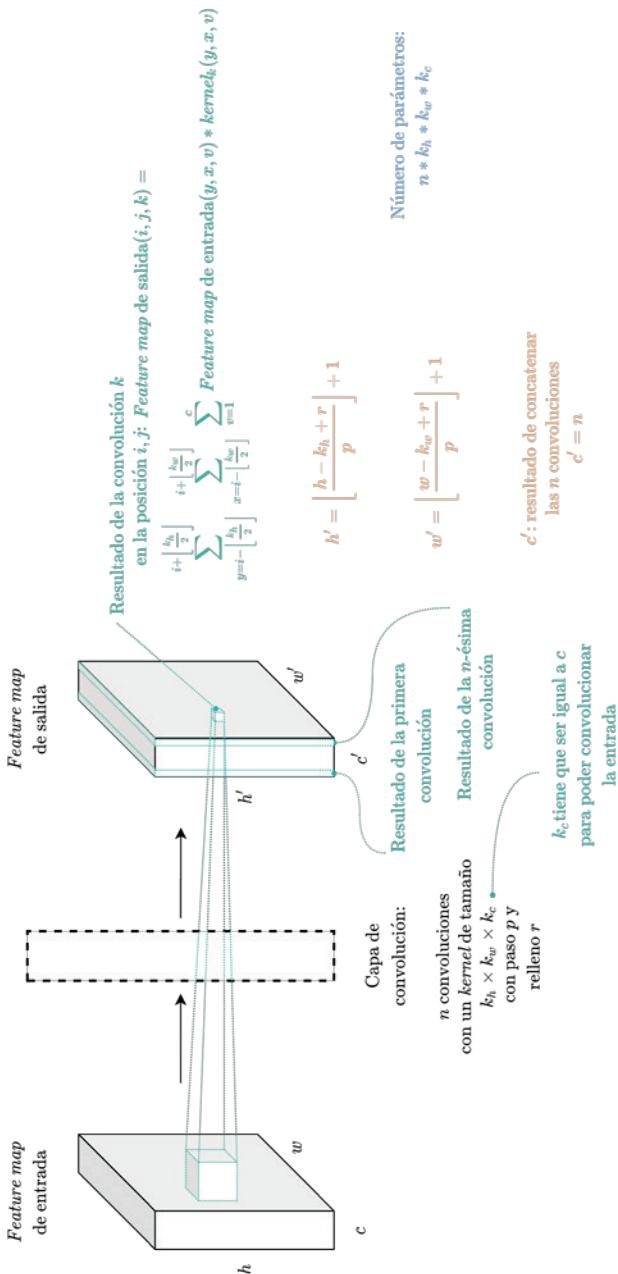
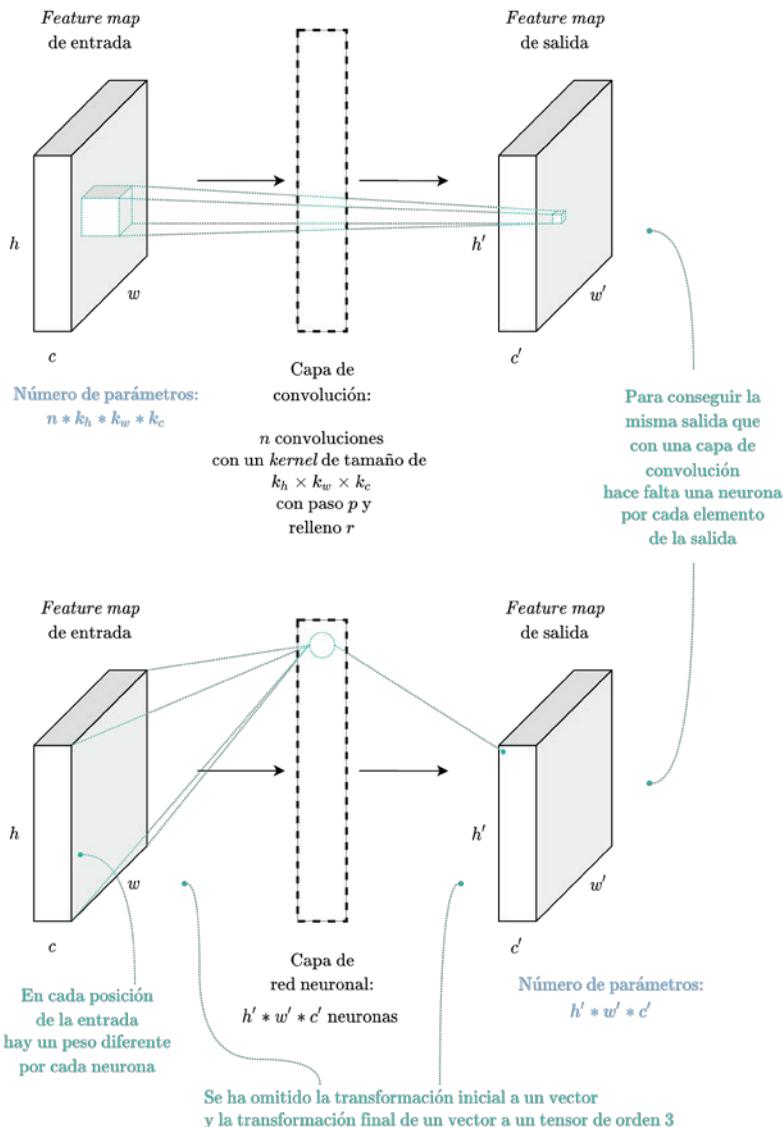
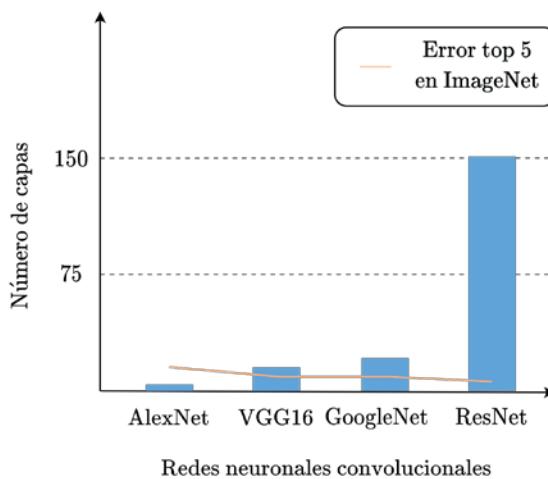


Figura 4.26: Información relativa a una capa de convolución



**Figura 4.27:** Comparativa entre una capa de una red neuronal y una capa de convolución



**Figura 4.28:** Mejora de los resultados con el aumento de la profundidad de la red. Error Top-5 es el error cometido por la red al no estar la clase esperada entre las 5 primeras clases del modelo.

### 4.2.1.2. Capa de *pooling*

Una de las primeras aplicaciones de las redes neuronales convolucionales fueron los problemas de clasificación. En este tipo de problemas las capas de convolución permiten detectar el elemento de interés en la entrada sin importar la posición que ocupe. En el caso de las imágenes, si, por ejemplo, se desea clasificar imágenes de dígitos escritos a mano, no se necesita conocer la posición del dígito en la imagen sino saber la presencia o no de uno u otro dígito.

Con la convolución se pueden detectar características en cualquier posición, sin embargo, la característica que se quiere detectar puede variar ligeramente. Por ejemplo, un borde no siempre es completamente recto, pero se pretende seguir detectándolo a pesar de esta pequeña variación. La capa de *pooling* permite dotar de cierta invarianza a pequeñas deformaciones de las características mediante la reducción del tamaño de los datos tomando algún valor representativo de una región. Además, al no contar con parámetros ajustables y al reducir el tamaño de los datos, reduce el número de parámetros que tiene que ajustar la red y aumenta el campo receptivo de las capas de convolución posteriores.

La operación de *pooling* es parecida a la convolución: se realiza una determinada operación de forma local sobre cada posición de la entrada. Es por ello que cuenta con el mismo tipo de parámetros que la capa de convolución. Estos parámetros son el tamaño de la ventana, el paso y el relleno.

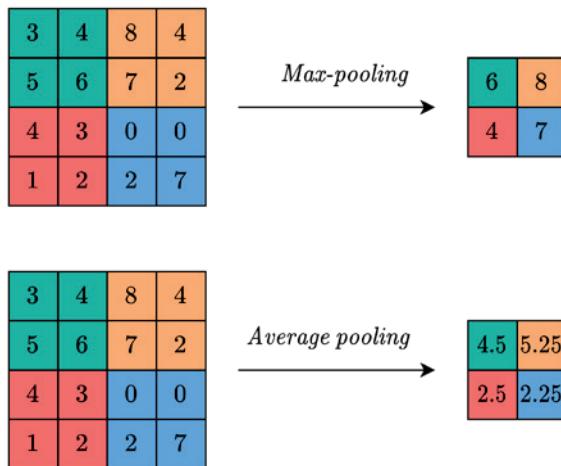
El tamaño de ventana es el equivalente al tamaño del *kernel* de convolución de una capa de convolución. Indica la región de la entrada que se procesa en cada posición. El paso va a permitir reducir el tamaño de la entrada y es por ello que es normal utilizar un paso mayor a uno. Por último, el relleno se utiliza para mantener un tamaño proporcional a los datos de entrada en la salida.

El cálculo de la salida de una capa de *pooling* depende del tipo de operación elegida, normalmente suele tratarse de un valor representativo de la región que se analiza. Las operaciones más utilizadas son:

- *Max-pooling*: asigna como salida en la posición actual el máximo valor de la región analizada.
- *Average pooling*: asigna como salida en la posición actual la media de la región analizada.

En la Figura 4.29 se muestra un ejemplo de la operación de *max-pooling* y *average*

*pooling* sobre una entrada de ejemplo. Como se observa en la figura, independientemente del tipo de operación que se utiliza, en cada posición de la salida el valor obtenido no varía por mover de posición los datos dentro de una ventana de entrada, lo que permite ganar, dentro de esta ventana, una invariancia a la estructura espacial de los datos.



**Figura 4.29:** Ejemplo del cálculo de *max-pooling* y *average pooling*

En el caso de las redes neuronales convolucionales, la capa de *pooling* procesa cada uno de los canales del mapa de características individualmente y se concatenan al finalizar para dar la salida.

### 4.2.1.3. Conexiones

Las primeras redes neuronales convolucionales seguían un patrón de capas que consistía en repetir una secuencia de capas de convolución y *pooling*. El mapa de características de la última capa era procesado por una red neuronal que lo transforma a la representación deseada de salida.

En este patrón repetitivo, los mapas de características son procesados de forma se-

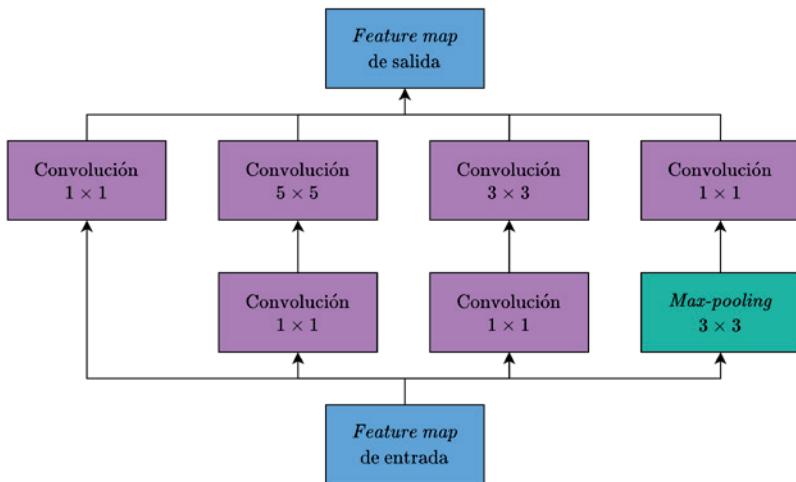
cuencial por cada capa. Las primeras mejoras que se consiguieron en las redes neuronales convolucionales estaban relacionadas con el aumento de la profundidad de estas, es decir, del número de capas de la etapa de extracción de características. Estas redes se centraban en seguir repitiendo la combinación de capas de convolución y *pooling* con una elección más refinada de sus parámetros. Esta forma de hacer más profunda la arquitectura provoca un aumento del número de parámetro con cada capa que se introduce. Además del empleo de más capas de convolución, el número de convoluciones que se realiza en cada capa se suele aumentar con la profundidad de la red. Esto es debido a que se busca extraer más características y darle más libertad al modelo. Sin embargo, esto provoca un mayor número de parámetros, tanto para la capa que procesa los datos como para la siguiente capa, que tendrá más canales de entrada que procesar.

Con el objetivo de mejorar los resultados de la red sin necesidad de aumentar de forma notoria el número de parámetros, se comenzó a explorar alternativas a la consecución de capas de convolución y *pooling*. Se buscaba combinar estas operaciones de forma que mejoren el desempeño de la red sin necesidad de introducir más parámetros, utilizándolas de una forma más «inteligente» que la simple repetición de las mismas operaciones capa tras capa.

A continuación, se detallan las principales alternativas de conexiones entre las capas de una red neuronal convolucional.

**Secuencial.** Este tipo de conexión es la tradicional. Se emplean capas de convolución seguidas de una capa de *pooling* para reducir el tamaño de los datos, ganar cierta invarianza a pequeñas deformaciones y procesar una región más grande de la red sin aumentar el número de parámetros.

**Inception block.** Este fue uno de los primeros enfoques diferente al tradicional. Para un mismo mapa de características de entrada se emplean diferentes combinaciones de operaciones y/o parámetros para estas operaciones (como distintos tamaños del *kernel* de convolución p. ej.). En la Figura 4.30 se muestra un ejemplo de bloque *inception* donde se observa cómo el mismo mapa de características es procesado con diferentes operaciones y, después, es combinado (mediante la concatenación del resultado de las diferentes operaciones) en un único mapa de características como si de la salida una única capa de convolución se tratase.

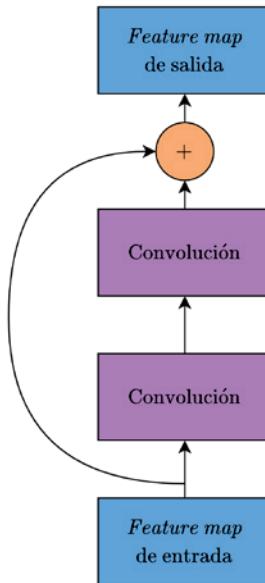


**Figura 4.30:** Ejemplo de bloque *inception*

**Skip connection.** Este tipo de conexiones surgieron a partir de un estudio realizado por los autores de la red ResNet [31]. Estos autores se percataron de que, al añadir más capas, no lograban bajar el error que la red cometía durante el entrenamiento. Esto parece contraintuitivo, ya que, si una red con menos capas consigue cierto error, es de esperar que una red que use más capas logre, al menos, el mismo error. Esto podría ocurrir si, llegado a cierto número de capas, no es necesario una representación más compleja de los datos, por lo que es de esperar que la red aprenda, en las capas que les son innecesarias, a realizar una transformación identidad para no modificar la representación obtenida en capas anteriores. Sin embargo, los investigadores comprobaron que para la red es bastante difícil aprender la transformación identidad, por lo que optaron por darle una forma más sencilla de poderla realizar si la necesita.

Las *skip connections* se propusieron como solución a este problema (ver Figura 4.31). Esta conexión consiste en sumar la salida de una capa con la entrada de alguna capa anterior. Es decir, siendo  $x$  la entrada y  $f$  la transformación que se realiza en la capa, la salida de la capa sería  $f(x) + x$ . En caso de necesitar la transformación

identidad, solo se necesitaría poner los pesos de la capa a cero y la salida de esta será la entrada.

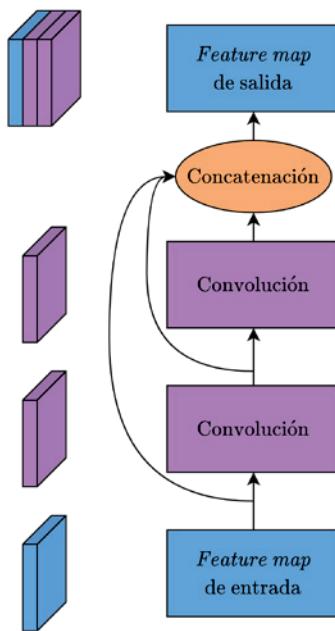


**Figura 4.31:** Ejemplo de *skip connection*

Además de corregir el problema de entrenar redes muy profundas, al añadir la entrada anterior mediante una operación de adición, se logra contrarrestar el problema del *vanish gradient*, ya que el gradiente que se propaga hacia las primeras capas no se multiplica por ningún valor.

**Dense block.** El principal inconveniente del uso de *skip connections* es que tanto la información de la capa anterior como la de la actual no está disponible de forma individual. Para mantener la información anterior sin perder la nueva, los *dense blocks* concatenan mapas de características de capas anteriores en único mapa de características (ver Figura 4.32).

Utilizando los *dense blocks*, la red puede añadir nueva información sin perder nunca la información anterior. Sin embargo, esta estrategia tiene la desventaja de que



**Figura 4.32:** Ejemplo de *dense block*

hace a las convoluciones de las capas usar más parámetros al haber más canales de entrada.

## 4.2.2. Aplicaciones

Los resultados conseguidos por las redes neuronales convolucionales propiciaron su aplicación a multitud de campos diferentes y el surgimiento de un ecosistema empresarial alrededor del uso de técnicas de *deep learning*. Este ecosistema empresarial ha propiciado la adopción de las técnicas de *deep learning* para aplicaciones tales como la búsqueda de nuevos fármacos, detección de enfermedades, conducción autónoma y un largo etcétera. En la Figura 4.33 se encuentran alguno de los casos de uso más comunes.

Las aplicaciones de las redes neuronales convolucionales a diferentes campos de aplicación han permitido superar los resultados de las técnicas del estado del arte. Desde el procesamiento de imágenes, pasando por el procesamiento de audio o texto, las redes neuronales convolucionales han demostrado su potencialidad. Por ejemplo, su aplicación a la medicina o la agricultura ha permitido el desarrollo de multitud de nuevas empresas que buscan automatizar muchas de las tareas que en estos campos se realiza.

Para ser aplicadas a campos tan diversos, ha sido necesario desarrollar nuevos modelos que sean capaces de tratar con las peculiaridades de cada problema. Cuando no es posible replantear un problema para que una red neuronal convolucional pueda abordarlo con buenos resultados es necesario modificar su arquitectura. La mayoría de las modificaciones se basan en añadir operaciones diferentes a la propia red o añadirle una componente temporal para facilitar el tratamiento de este tipo de datos.

En lo que al campo de procesamiento de imágenes se refiere, las redes neuronales convolucionales se han convertido en la técnica de referencia para abordar problemas de clasificación, segmentación o detección en imágenes gracias a sus resultados (ver Figura 4.34). Empresas como Tesla hacen uso de estas redes para desarrollar su sistema de conducción autónoma. Las redes neuronales convolucionales procesan las imágenes que toman las diferentes cámaras del coche para detectar los elementos de interés (como personas o señales de tráfico) del entorno y, en última instancia, conducir el vehículo. En la Figura 4.35 se muestra información sobre la red de Tesla, HydraNet.

Algunos problemas de procesamiento de imágenes han requerido del uso de redes neuronales recurrentes para, por ejemplo, describir una imagen. Un modelo de

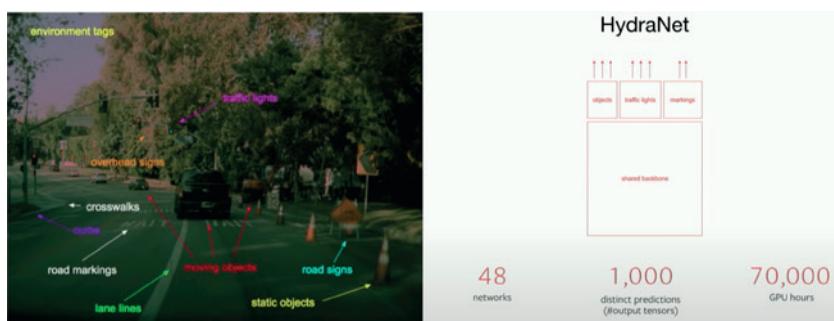
General use case	Industry
<b>Sound</b>	
Voice recognition	UX/UI, Automotive, Security, IoT
Voice search	Handset maker, Telecoms
Sentiment analysis	CRM
Flaw detection (engine noise)	Automotive, Aviation
Fraud detection (latent audio artifacts)	Finance, Credit Cards
<b>Time Series</b>	
Log analysis/Risk detection	Data centers, Security, Finance
Enterprise resource planning	Manufacturing, Auto., Supply chain
Predictive analysis using sensor data	IoT, Smart home, Hardware manufact.
Business and Economic analytics	Finance, Accounting, Government
Recommendation engine	E-commerce, Media, Social Networks
<b>Text</b>	
Sentiment Analysis	CRM, Social media, Reputation mgt.
Augmented search, Theme detection	Finance
Threat detection	Social media, Govt.
Fraud detection	Insurance, Finance
<b>Image</b>	
Facial recognition	
Image search	Social media
Machine vision	Automotive, aviation
Photo clustering	Telecom, Handset makers
<b>Video</b>	
Motion detection	Gaming, UX, UI
Real-time threat detection	Security, Airports

**Figura 4.33:** Principales casos de uso del *deep learning*Fuente: [https://deeplearning4j.org/img/use\\_case\\_industries.png](https://deeplearning4j.org/img/use_case_industries.png)



**Figura 4.34:** De izquierda a derecha: clasificación, detección y segmentación

Fuente: <https://code.facebook.com/posts/561187904071636>



**Figura 4.35:** Red de conducción autónoma de Tesla: HydraNet

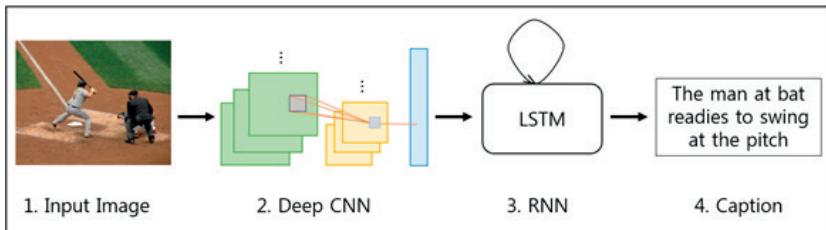
Fuente:

<https://phucnsp.github.io/blog/self-taught/2020/04/30/tesla-nn-in-production.html>

referencia dentro de las redes recurrentes son las *long short term memory* (LSTM) [32], que combinadas con redes neuronales convolucionales para extraer características, son capaces de hacer una descripción del contenido de una imagen [33] como se muestra en la Figura 4.36.

Sin necesidad de ser combinadas con otros modelos como las LSTM, se ha logrado desarrollar modelos derivados directamente de las redes neuronales convolucionales que pueden aplicarse a problemas de elevada complejidad. Es el caso de los *autoencoders* [34] y las *generative adversarial networks* [35].

Los *autoencoders* se encargan de transformar los datos de entrada a una representación mucho más reducida que permita su recuperación. Esto hace que, al tener que representar la entrada en una dimensión mucho menor, se puedan conseguir robustez en la representación de los datos, logrando invarianza a multitud de trans-



**Figura 4.36:** Red neuronal convolucional y LSTM usadas para describir una imagen

Fuente: <http://brain.kaist.ac.kr/research.html>

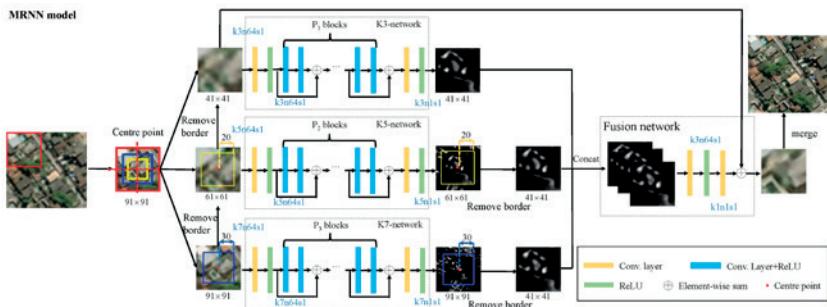
formaciones o ruido y es capaz de agrupar objetos similares en la misma representación.

Por su parte, las *generative adversarial networks* (GAN) son dos redes neuronales convolucionales cuyos modelos son ajustados cumpliendo objetivos opuestos: por un lado, una de las redes intenta discernir si su entrada viene de la otra red o se trata de un dato del conjunto de datos; y, por otro lado, la otra red intenta «engaños» a su homóloga. Gracias a esta forma de combinar las redes se puede conseguir lo que se llama una *red generadora*, capaz, a partir de entradas aleatorias, de crear datos como los presentes en el conjunto de datos de entrenamiento.

Con estos dos modelos se pueden conseguir aplicaciones sorprendentes como las que se muestran en las Figuras 4.37, 4.38, 4.39, 4.40, 4.41 y 4.42.

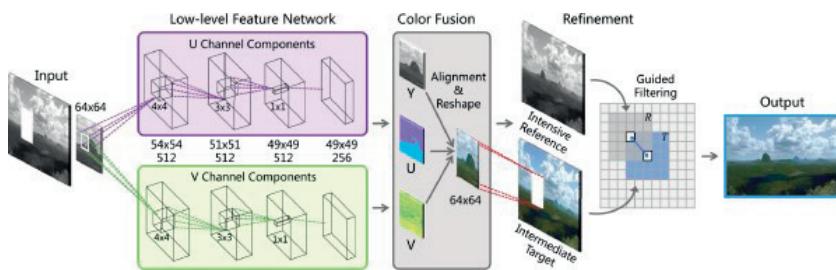
No solo se limitan a trabajar con problemas estáticos. Al tener tan buen desempeño extrayendo características de forma automática, las redes neuronales convolucionales se han empleado en problemas de aprendizaje por refuerzo en entornos simulados como pueden ser los videojuegos. Un ejemplo de este uso son los modelos creados para jugar a juegos de Atari [36] (ver Figura 4.43) o al juego de mesa Go, considerado un juego inabordable a través de técnicas de inteligencia artificial y que, en 2016, un modelo creado por DeepMind consiguió batir al campeón mundial de Go [37] (ver Figura 4.44). También se aplican en entornos reales, como puede ser la conducción autónoma [38] (ver Figura 4.45) o la manipulación de objetos por parte de brazos robóticos [39] (ver Figura 4.46).

Todas estas aplicaciones son posibles gracias a la combinación de algoritmos exis-



**Figura 4.37:** Reescalado de imágenes

Fuente: <https://www.mdpi.com/2072-4292/11/13/1588>

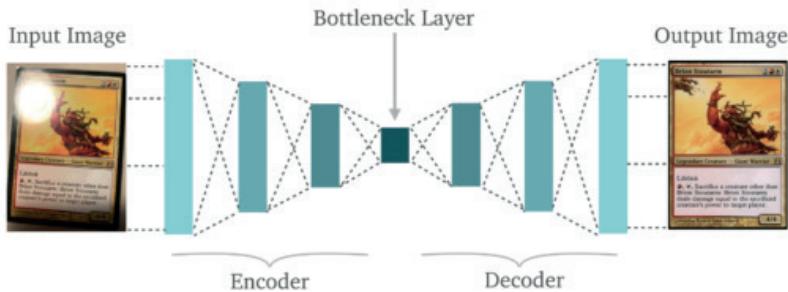


**Figura 4.38:** Reconstrucción y coloreado de imágenes

Fuente: <https://www.sciencedirect.com/science/article/pii/S0925231218306672>

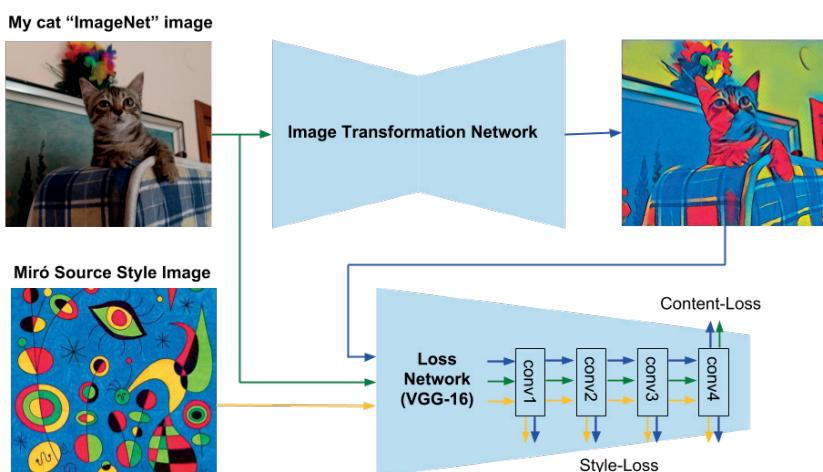
tentes con modificaciones, tanto de los modelos de *deep learning* originales como de los datos, para que puedan ser tratados mediante estas técnicas. Debido a su gran capacidad para extraer características de forma automática, estas técnicas se han trasladado al procesamiento del lenguaje [40] (ver Figura 4.47) e incluso al procesamiento de audio [41] (ver Figura 4.48). La combinación de estas aplicaciones puede utilizarse para construir sistemas complejos de traducción, tanto de texto, audio o combinaciones de ambos [42].

Estas transformaciones tienen que ver, en la mayoría de los casos, con modificar la salida del modelo de modo que sea capaz de tratar el problema actual. Por ejemplo, las redes neuronales convolucionales fueron diseñadas para realizar clasificación de



**Figura 4.39:** Representación invariante de objetos

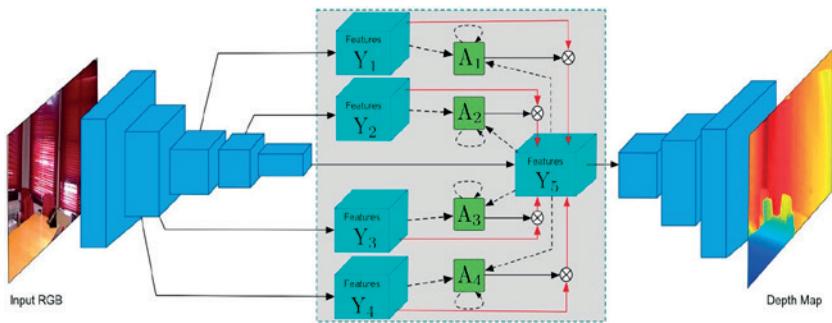
Fuente: <https://hackernoon.com/a-deep-convolutional-denoising-autoencoder-for-image-classification-26c777d3b88e>



**Figura 4.40:** Transferencia del estilo de una imagen a otra

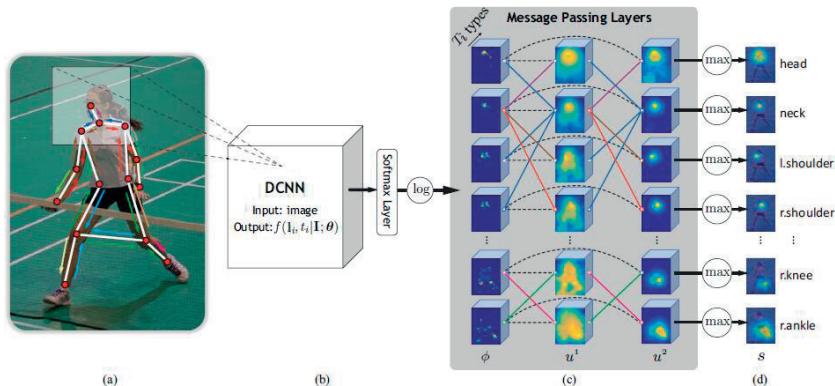
Fuente: [https://gombru.github.io/2019/01/14/miro\\_styletransfer\\_deeppdream/](https://gombru.github.io/2019/01/14/miro_styletransfer_deeppdream/)

imágenes y, por lo tanto, deben ser modificadas para poder realizar tareas tales como la segmentación por instancias. La segmentación por instancias permite clasificar los píxeles de una imagen en base a tanto la clase como al individuo al que pertenecen.



**Figura 4.41:** Estimación de la profundidad

Fuente: <http://www.robots.ox.ac.uk/~danxu/>

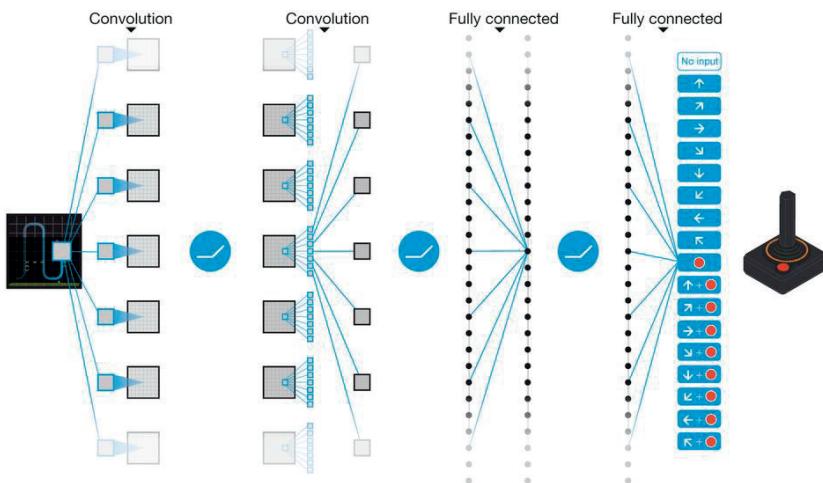


**Figura 4.42:** Estimación de la pose

Fuente:

[http://www.fubin.org/research/Human\\_Pose\\_Estimation/Human\\_Pose\\_Estimation.html](http://www.fubin.org/research/Human_Pose_Estimation/Human_Pose_Estimation.html)

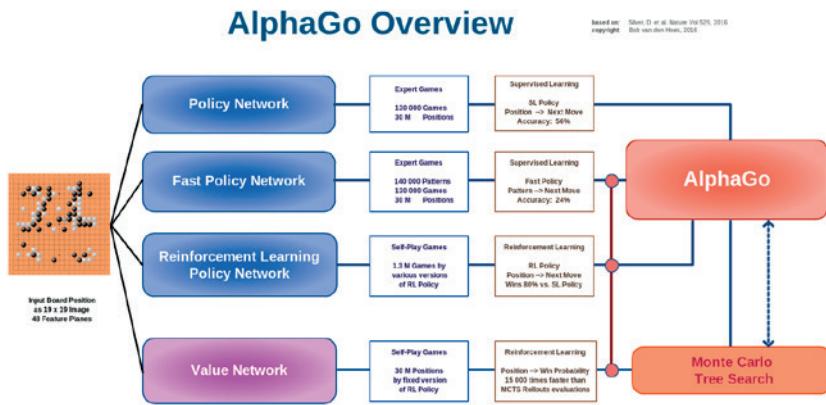
Existen diferentes enfoques para adaptar una red neuronal convolucional a realizar segmentación por instancias. El enfoque más utilizado consiste en hacer un proceso de detección con una posterior etapa de segmentación del individuo principal de cada una de las detecciones. El ejemplo más representativo de esta técnica es la red Mask R-CNN [43], presentada por Kaiming He y su equipo del laboratorio de inteligencia artificial de Facebook en 2017. En la Figura 4.49 se encuentran



**Figura 4.43:** Modelo basado en *deep learning* entrenado por aprendizaje por refuerzo para jugar a los juegos de Atari

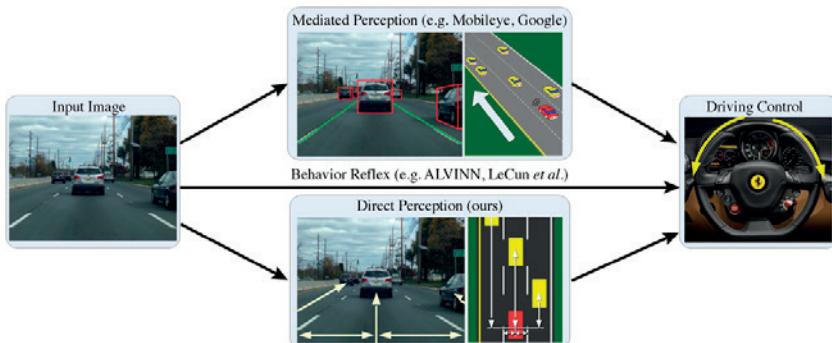
Fuente: <https://www.nature.com/articles/nature14236>

algunos ejemplos de las segmentaciones por instancias que se consiguen con esta red.



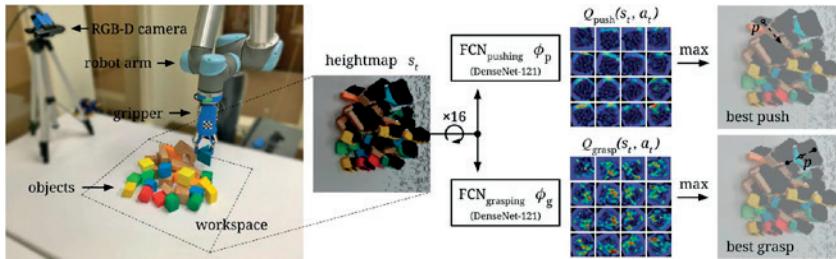
**Figura 4.44:** Modelo basado en *deep learning* entrenado por aprendizaje por refuerzo para jugar al juego del Go

Fuente: <http://deeplearningskysthelimit.blogspot.com/2016/04/part-2-alphago-under-magnifying-glass.html>



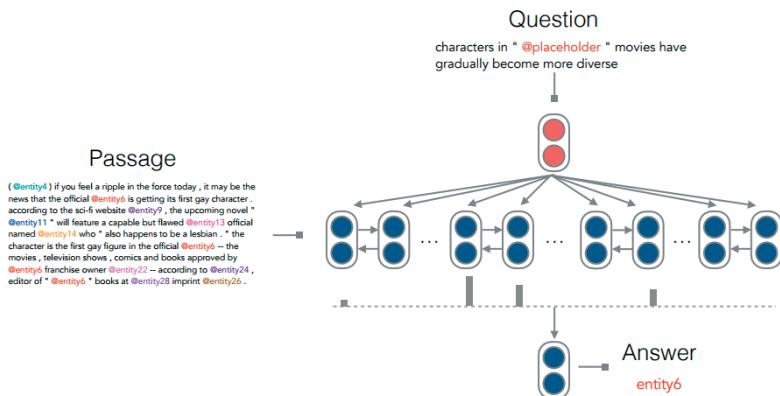
**Figura 4.45:** Conducción autónoma con modelo de *deep learning* entrenado con aprendizaje por refuerzo

Fuente: <http://deepdriving.cs.princeton.edu/>



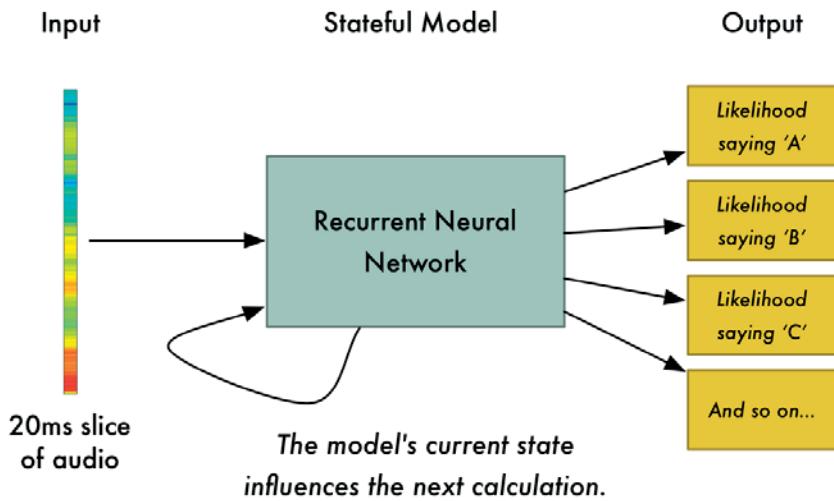
**Figura 4.46:** Manipulación de objetos con modelo de *deep learning* entrenado por aprendizaje por refuerzo

Fuente: <http://vpg.cs.princeton.edu/>



**Figura 4.47:** Procesamiento del lenguaje haciendo uso de una red neuronal convolucional

Fuente: <https://arxiv.org/pdf/1606.02858.pdf>



**Figura 4.48:** Reconocimiento del audio haciendo uso de una red neuronal convolucional combinada con una red neuronal recurrente

Fuente: <https://www.machinelearningisfun.com/posts>



**Figura 4.49:** Ejemplos de segmentaciones por instancias realizadas con la red Mask R-CNN

Fuente: <https://lilianweng.github.io/lil-log/assets/images/mask-rcnn-examples.png>

### 4.2.3. Arquitecturas populares

Cuando se aborda un problema desde la perspectiva de las redes neuronales convolucionales puede ahorrarse algo de tiempo si, en lugar de tener que diseñar la arquitectura a través de un proceso de prueba y error, se utiliza alguna arquitectura ya diseñada.

Existen arquitecturas que fueron diseñadas originalmente para competiciones muy exigentes. Gracias a los resultados conseguidos en estas competiciones, muchas de estas arquitecturas han quedado como referencia a la hora de abordar un problema con una red neuronal convolucional.

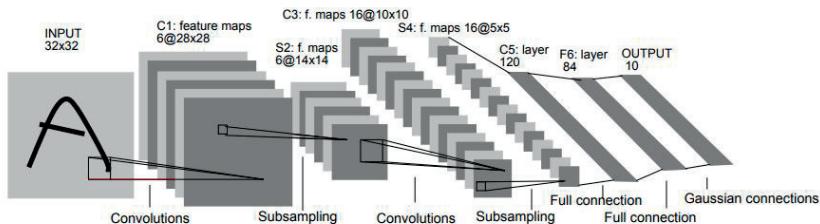
Al igual que en las redes neuronales, la estrategia del *transfer learning* es igualmente empleada en las redes neuronales convolucionales. Se puede utilizar alguna arquitectura ya entrenada y entrenar únicamente la red neuronal del final, aprovechando la extracción de características aprendida por la red en el problema que fue entrenada; o utilizarlo como punto de partida de un nuevo entrenamiento.

A continuación se detallan algunas de las principales arquitecturas que cualquier experto en *deep learning* debe conocer, ya sea por su repercusión histórica como por sus resultados y novedades que introdujeron. Hoy día, casi cualquier librería de *deep learning* suele incluir estas arquitecturas ya implementadas y listas para su uso.

#### 4.2.3.1. LeNet-5

La red LeNet-5 [28] es considerada la primera red neuronal convolucional de la historia. Fue presentada en 1998 por Yann LeCun y su equipo de investigación de los laboratorios Bell. Esta red fue desarrollada con el objetivo de reconocer dígitos escritos a mano para el servicio postal estadounidense.

En la Figura 4.50 se muestra la arquitectura de esta red. Se compone de dos capas de convolución y de *pooling* que se encuentran intercaladas. La salida de la última capa de *pooling* sirve como entrada de una red neuronal tradicional que cuenta con tres capas.



**Figura 4.50:** Arquitectura de la red LeNet-5

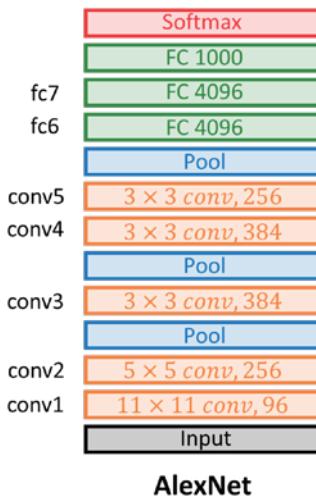
Fuente: <http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf>

#### 4.2.3.2. AlexNet

AlexNet [20] es el nombre de la primera red neuronal convolucional en ganar la competición ImageNet. Al superar por un amplio margen a la segunda metodología, supuso el comienzo de la popularización de las técnicas de *deep learning* para abordar la mayoría de problemas relacionados con el procesamiento de imágenes.

Esta red fue presentada en 2012 por Alex Krizhevsky bajo la tutela de Geoffrey Hinton. Su arquitectura (ver Figura 4.51) es más grande que su predecesora, LeNet-5. Cuenta con seis capas de convolución, dos capas de *pooling* y una red neuronal de cuatro capas. Al aumentar el tamaño de la red, también aumenta el número de parámetros a ajustar. En el caso de AlexNet, el número de parámetros asciende a más de 60 millones.

Otra de las novedades que introdujo AlexNet fue el uso de las GPUs para el entrenamiento de la red, lo que redujo el tiempo de entrenamiento de manera considerable. También popularizó el uso de la función de activación ReLU en las capas de convolución.



**Figura 4.51:** Arquitectura de la red AlexNet

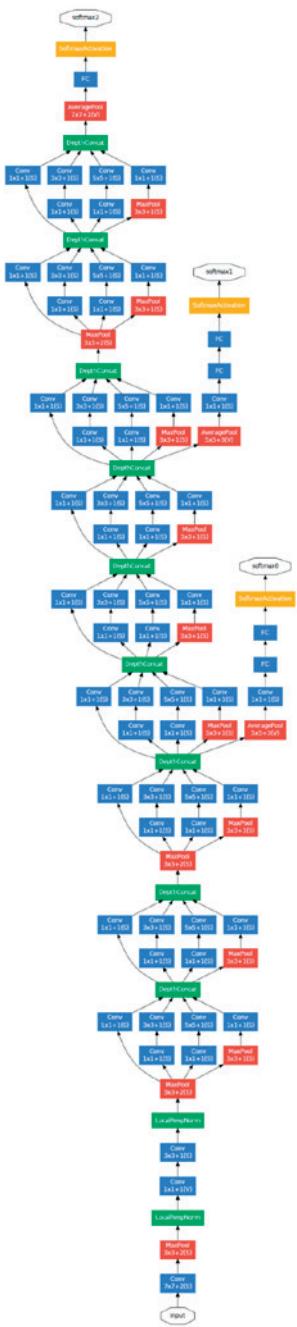
Fuente: <http://datahacker.rs/deep-learning-alexnet-architecture/>

### 4.2.3.3. GoogleNet

En 2014, la red ganadora de la competición ImageNet fue la red GoogleNet [44]. Esta red fue la que introdujo los *inception blocks* y se adentró en el aumento de la profundidad de las redes neuronales convolucionales para mejorar los resultados.

En la Figura 4.52 se muestra su arquitectura. Se observa el aumento del número de capas respecto a redes anteriores como AlexNet. Otra de las novedades que introduce GoogleNet es el uso de convoluciones con tamaño de *kernel*  $1 \times 1$  para reducir la dimensión de los datos y utilizar menos parámetros. La convolución  $1 \times 1$  actúa como un cuello de botella que obliga a la red a aprender a comprimir la información.

GoogleNet se compone de 22 capas, pero, gracias a las mejoras introducidas, el número de parámetros se reduce a solo 7 millones.



**Figura 4.52:** Arquitectura de la red GoogleNet

Fuente: <https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvrc-2014-image-classification-c2b3565a64e7>

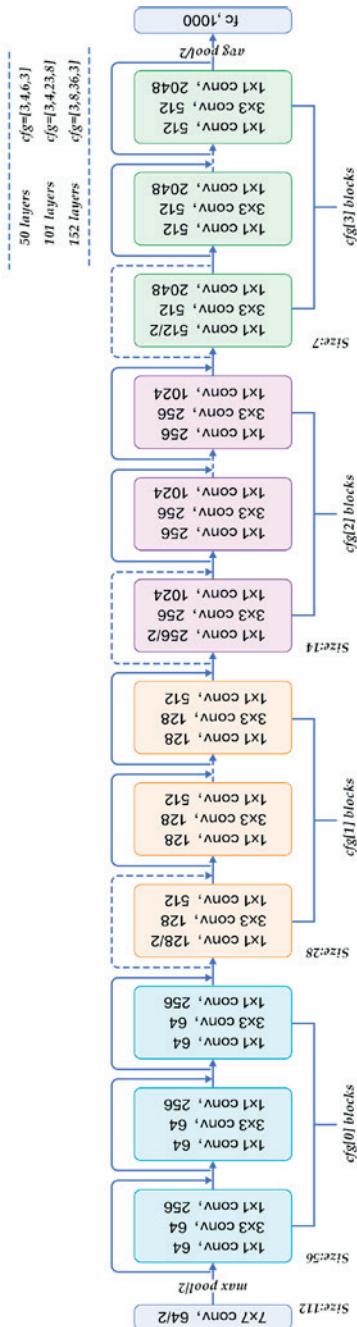
#### 4.2.3.4. ResNet

Al igual que otras, la red ResNet [31] se popularizó a raíz de los resultados conseguidos en la competición ImageNet, donde se convirtió en la ganadora en el año 2015.

Su principal aportación fue las *skip connections*. Además, propuso el uso de un *average pooling* global (con el mismo tamaño de ventana que el que tenían los datos) para procesar el último mapa de características y, así, transformarlo en un vector en el que se recoge el valor medio de cada canal del mapa de características (lo que reduce el número de pesos que utiliza la red neuronal del final para generar la salida). También fue de las primeras redes en hacer uso de *batch normalization* a la salida de las capas de convolución.

Esta red se presentó con diferentes variantes de la profundidad de la red. Concretamente, se presentaron las redes ResNet-18, ResNet-50, ResNet-101 y ResNet-152 en referencia al número de bloques de cada una. En la Figura 4.53 se muestra la arquitectura de las redes ResNet-50, ResNet-101 y ResNet-152.

A pesar de contar con hasta 152 bloques, la red ResNet-152 cuenta solo con 11 millones de parámetros y es la primera en bajar del considerado error humano (en torno al 5 %) en la competición ImageNet.

**Figura 4.53:** Arquitectura de las redes ResNet-50, ResNet-101 y ResNet-152Fuente: [https://miro.medium.com/max/2800/0\\*pkrs08DZa0m6IAcJ.png](https://miro.medium.com/max/2800/0*pkrs08DZa0m6IAcJ.png)

#### 4.2.3.5. DenseNet

Después de la llegada de las *skip connections*, la red DenseNet [45], ganadora de la competición ImageNet en 2016, presentó los *dense blocks*. Al preserva la información de entradas anteriores, la red consiguió mejorar los resultados de sus predecesoras.

Gracias al diseño de su arquitectura, a pesar de aumentar considerablemente la profundidad de la red, la versión más grande de la red, DenseNet-169, solo utiliza 12 millones de parámetros para 169 capas. En la Figura 4.54 se muestra la arquitectura de la red.

Como contrapartida, esta red requiere de más capacidad de procesamiento al realizar muchas más operaciones que otras arquitecturas alternativas y manejar mapas de características de mayor tamaño.

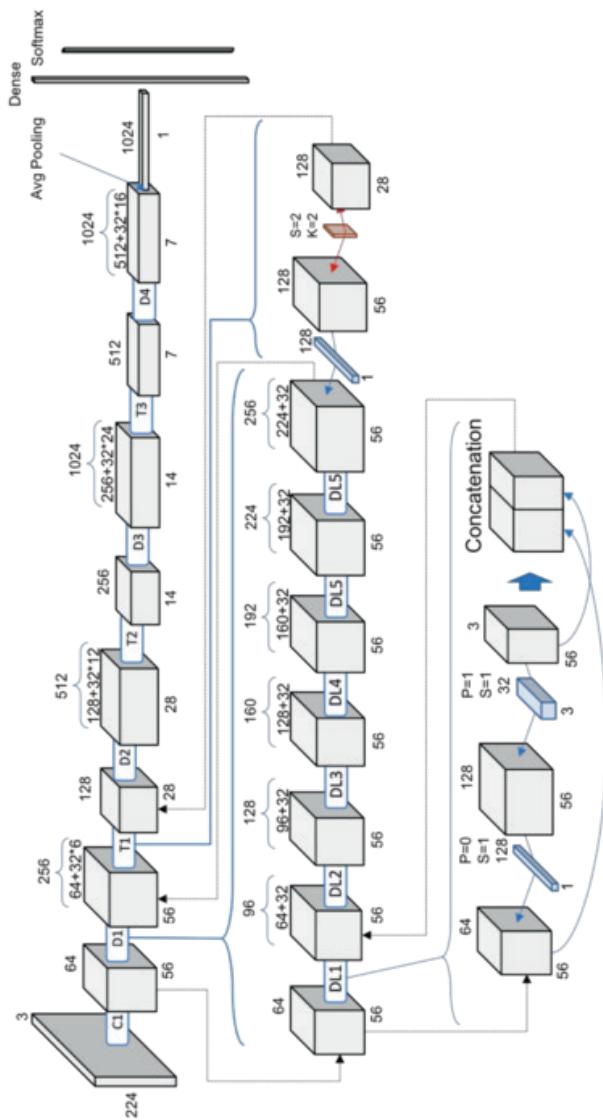


Figura 4.54: Arquitectura de la red DenseNet-121

Fuente: <https://towardsdatascience.com/understanding-and-visualizing-densenets-7f688092391a>

# Bibliografía

- [1] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.
- [2] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.
- [3] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.,” *Psychological review*, vol. 65, no. 6, p. 386, 1958.
- [4] B. Widrow and M. E. Hoff, “Adaptive switching circuits,” tech. rep., Stanford Univ Ca Stanford Electronics Labs, 1960.
- [5] K. Hornik, “Approximation capabilities of multilayer feedforward networks,” *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.
- [6] J. Su, D. V. Vargas, and K. Sakurai, “One pixel attack for fooling deep neural networks,” *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 5, pp. 828–841, 2019.
- [7] D. O. Hebb, *The organization of behavior: a neuropsychological theory*. J. Wiley; Chapman & Hall, 1949.
- [8] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.
- [9] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

- [10] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *J. Mach. Learn. Res.*, vol. 12, p. 2121–2159, July 2011.
- [11] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.
- [12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.
- [13] K. Kawaguchi, J. Huang, and L. P. Kaelbling, “Every local minimum value is the global minimum value of induced model in nonconvex machine learning,” *Neural Computation*, vol. 31, no. 12, pp. 2293–2323, 2019.
- [14] A. Choromanska, M. Henaff, M. Mathieu, G. Arous, and Y. LeCun, “The loss surfaces of multilayer networks. arxiv 2014,” *arXiv preprint arXiv:1412.0233*, 2015.
- [15] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, “Understanding deep learning requires rethinking generalization,” *arXiv preprint arXiv:1611.03530*, 2016.
- [16] D. H. Wolpert, “The lack of a priori distinctions between learning algorithms,” *Neural computation*, vol. 8, no. 7, pp. 1341–1390, 1996.
- [17] J. J. Hopfield, “Neural networks and physical systems with emergent collective computational abilities,” *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.
- [18] H. T. Siegelmann and E. D. Sontag, “On the computational power of neural nets,” *Journal of computer and system sciences*, vol. 50, no. 1, pp. 132–150, 1995.
- [19] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.
- [20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

- [21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in 2009 IEEE conference on computer vision and pattern recognition, pp. 248–255, Ieee, 2009.
- [22] C. Metz, “Turing award won by 3 pioneers in artificial intelligence,” The New York Times, Mar 2019.
- [23] “Artificial intelligence: Google’s alphago beats go master lee se-dol,” BBC, Mar 2016.
- [24] G. A. Montes and B. Goertzel, “Distributed, decentralized, and democratized artificial intelligence,” Technological Forecasting and Social Change, vol. 141, pp. 354–358, 2019.
- [25] E. Strubell, A. Ganesh, and A. McCallum, “Energy and policy considerations for deep learning in nlp,” arXiv preprint arXiv:1906.02243, 2019.
- [26] D. H. Hubel and T. N. Wiesel, “Receptive fields of single neurones in the cat’s striate cortex,” The Journal of physiology, vol. 148, no. 3, p. 574, 1959.
- [27] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” Biological Cybernetics, vol. 36, no. 4, pp. 193–202, 1980.
- [28] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.
- [29] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” in European conference on computer vision, pp. 818–833, Springer, 2014.
- [30] S. R. Kheradpisheh, M. Ghodrati, M. Ganjtabesh, and T. Masquelier, “Deep networks can resemble human feed-forward vision in invariant object recognition,” Scientific reports, vol. 6, p. 32672, 2016.
- [31] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778, 2016.
- [32] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.

- [33] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, “Show and tell: A neural image caption generator,” in Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 3156–3164, 2015.
- [34] P. Baldi, “Autoencoders, unsupervised learning, and deep architectures,” in Proceedings of ICML workshop on unsupervised and transfer learning, pp. 37–49, 2012.
- [35] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in Advances in neural information processing systems, pp. 2672–2680, 2014.
- [36] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” arXiv preprint arXiv:1312.5602, 2013.
- [37] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, “A general reinforcement learning algorithm that masters chess, shogi, and go through self-play,” Science, vol. 362, no. 6419, pp. 1140–1144, 2018.
- [38] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, “Deepdriving: Learning affordance for direct perception in autonomous driving,” in Proceedings of the IEEE International Conference on Computer Vision, pp. 2722–2730, 2015.
- [39] S. Gu, E. Holly, T. Lillicrap, and S. Levine, “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates,” in 2017 IEEE international conference on robotics and automation (ICRA), pp. 3389–3396, IEEE, 2017.
- [40] D. Chen, J. Bolton, and C. D. Manning, “A thorough examination of the cn-n/daily mail reading comprehension task,” arXiv preprint arXiv:1606.02858, 2016.
- [41] L. Deng, G. Hinton, and B. Kingsbury, “New types of deep neural network learning for speech recognition and related applications: An overview,” in 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 8599–8603, IEEE, 2013.

- [42] R. J. Weiss, J. Chorowski, N. Jaitly, Y. Wu, and Z. Chen, “Sequence-to-sequence models can directly translate foreign speech,” [arXiv preprint arXiv:1703.08581](#), 2017.
- [43] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask r-cnn,” in [Proceedings of the IEEE international conference on computer vision](#), pp. 2961–2969, 2017.
- [44] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in [Proceedings of the IEEE conference on computer vision and pattern recognition](#), pp. 1–9, 2015.
- [45] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in [Proceedings of the IEEE conference on computer vision and pattern recognition](#), pp. 4700–4708, 2017.





SE TERMINÓ DE EDI-  
TAR EL LIBRO “DEEP  
LEARNING” EN ABRIL  
DE 2021, ESTANDO AL  
CUIDADO DE LA EDICIÓN  
EL SERVICIO DE PUBLI-  
CACIONES DE LA UNI-  
VERSIDAD DE HUELVA

