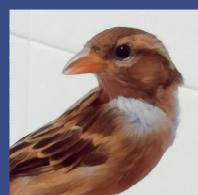
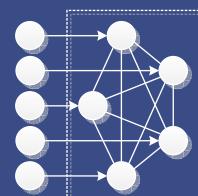
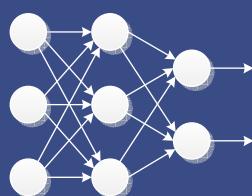


Fernando Berzal

REDES NEURONALES & DEEP LEARNING



REDES NEURONALES & DEEP LEARNING

FERNANDO BERZAL

REDES NEURONALES & DEEP LEARNING

GRANADA 2018

<http://deep-learning.ikor.org>

Copyright © 2018 Fernando Berzal

License information: LATEX Tufte-style book template licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Índice

PARTE I. REDES NEURONALES ARTIFICIALES 1

Inteligencia Artificial 3

Definiciones de I.A. 8

Los problemas de la I.A. 10

Técnicas heurísticas 22

Aprendizaje automático 27

Aplicaciones del aprendizaje automático 30

Tipos de aprendizaje automático 34

Aprendizaje supervisado 35

Sistemas de recuperación de información 38

Sistemas de pregunta-respuesta 39

Sistemas de recomendación 40

Nada es gratis 41

Aprendizaje no supervisado 42

Clustering 43

Reglas de asociación 49

¿Otras formas de aprendizaje? 51

Aprendizaje por refuerzo 52

Aprendizaje por asociación 54

Aprendizaje por imitación 56

Su uso en aprendizaje automático 57

<i>Evaluación de los resultados del aprendizaje</i>	59
<i>Métricas de evaluación</i>	60
<i>Precisión y tasa de error</i>	61
<i>Medidas sensibles a costes</i>	63
<i>Evaluación de rankings</i>	65
<i>Predicciones cuantitativas</i>	71
<i>Otras medidas</i>	76
<i>Métodos de evaluación</i>	79
<i>Conjuntos de entrenamiento, validación y prueba</i>	81
<i>Muestreo sin reemplazo: Validación cruzada</i>	84
<i>Muestreo con reemplazo: Bootstrapping</i>	86
<i>Finalización del modelo</i>	87
<i>Descomposición del error en sesgo y varianza</i>	88
<i>Técnicas de aprendizaje</i>	95
<i>Técnicas simbólicas: I.A. simbólica</i>	97
<i>Árboles de decisión</i>	98
<i>Inducción de reglas y listas de decisión</i>	101
<i>Técnicas analógicas: Reconocimiento de patrones</i>	102
<i>Vecinos más cercanos</i>	104
<i>Máquinas de vectores de soporte (SVM)</i>	108
<i>Técnicas bayesianas: Modelos probabilísticos</i>	110
<i>Naïve Bayes</i>	113
<i>Redes bayesianas</i>	114
<i>Redes de Markov</i>	116
<i>Técnicas evolutivas: Computación evolutiva</i>	117
<i>Algoritmos genéticos</i>	120
<i>Programación evolutiva</i>	122
<i>Estrategias de evolución</i>	123
<i>Programación genética</i>	124

<i>Técnicas conexionistas: I.A. conexionista</i>	125
<i>Combinación de múltiples modelos: Ensembles</i>	129
<i>Bagging</i>	131
<i>Boosting</i>	132
<i>Stacking</i>	133
<i>Selección y extracción de características</i>	134
<i>Selección de características</i>	136
<i>Extracción de características</i>	137
<i>Ingeniería de características</i>	140
<i>Deep Learning</i>	141
<i>Características clave del deep learning</i>	145
<i>Limitaciones del deep learning</i>	153
<i>Problemas derivados del sobreaprendizaje</i>	154
<i>Las redes neuronales como cajas negras</i>	156
<i>Inspiración biológica</i>	160
<i>Plasticidad</i>	163
<i>Organización jerárquica</i>	166
<i>Percepción</i>	170
<i>Modelos de neuronas y redes neuronales artificiales</i>	185
<i>Neuronas o elementos de procesamiento</i>	187
<i>Funciones de activación</i>	194
<i>Función de activación lineal</i>	195
<i>Función escalón</i>	196
<i>Función lineal con saturación</i>	196
<i>Función de activación sigmoidal</i>	197
<i>Función de activación lineal rectificada</i>	201
<i>Funciones de base radial</i>	203

<i>Arquitecturas de las redes neuronales artificiales</i>	204
<i>Redes feed-forward</i>	204
<i>Redes competitivas</i>	206
<i>Redes recurrentes</i>	207
<i>Simulación de neuronas biológicas</i>	210
<i>PARTE II. ENTRENAMIENTO DE REDES</i>	213
<i>Perceptrones</i>	215
<i>La neurona de McCulloch y Pitts</i>	215
<i>Modelo computacional de McCulloch y Pitts</i>	216
<i>Modelo neuronal de McCulloch y Pitts</i>	217
<i>Ejemplo: Percepción fisiológica del calor y del frío</i>	220
<i>Reconocimiento de patrones con redes neuronales</i>	221
<i>El algoritmo de aprendizaje del perceptrón</i>	226
<i>Interpretación geométrica</i>	229
<i>Corrección del algoritmo de aprendizaje del perceptrón</i>	231
<i>Variantes y aplicaciones del perceptrón</i>	235
<i>Actualizaciones de los pesos</i>	236
<i>Tasas de aprendizaje</i>	236
<i>Zonas muertas</i>	237
<i>El perceptrón sigmoidal: Entradas y salidas continuas</i>	239
<i>El algoritmo del bolsillo: Clases no linealmente separables</i>	240
<i>El perceptrón de estabilidad óptima</i>	242
<i>El perceptrón multiclasa</i>	243
<i>El perceptrón promedio</i>	245
<i>El perceptrón estructurado</i>	248
<i>Limitaciones del perceptrón</i>	251
<i>La función XOR</i>	252
<i>¿Son dos bits iguales?</i>	253
<i>¿Tienen dos patrones el mismo número de bits?</i>	254
<i>El libro de Minsky y Papert</i>	255
<i>Epílogo: Cómo resolver el problema XOR con perceptrones</i>	256

<i>El algoritmo de propagación de errores</i>	259
<i>Entrenamiento de redes multicapa</i>	263
<i>Entrenamiento de neuronas lineales</i>	264
<i>Un ejemplo intuitivo</i>	268
<i>La regla delta</i>	272
<i>Entrenamiento de neuronas sigmoidales</i>	278
<i>Entrenamiento de neuronas ocultas</i>	282
<i>El algoritmo de propagación de errores hacia atrás</i>	285
<i>Recapitulación, a modo de receta</i>	290
<i>El gradiente en las distintas capas de la red</i>	291
<i>Implementación del algoritmo</i>	294
<i>Funciones de activación</i>	294
<i>Propagación hacia adelante</i>	295
<i>Función de error</i>	296
<i>Propagación hacia atrás</i>	297
<i>Actualización de los pesos de la red</i>	299
<i>El proceso de entrenamiento</i>	302
<i>Inicialización de los pesos</i>	304
<i>Preprocesamiento de los datos de entrada</i>	305
<i>Presentación de los datos de entrada</i>	306
<i>Hiperparámetros del algoritmo</i>	307
<i>Depuración de la implementación</i>	308
<i>Sobre el cálculo del gradiente</i>	309
<i>Implementación modular</i>	310
<i>Implementación en una GPU</i>	313
<i>Variaciones sobre el tema</i>	314
<i>Coda: Un poco de historia</i>	315
<i>Entrenamiento de una red neuronal</i>	323
<i>Topología de la red</i>	326
<i>Aproximadores universales</i>	327
<i>Complejidad del aprendizaje</i>	330
<i>Tamaño de la red</i>	334
<i>Profundidad de la red</i>	336
<i>Conectividad de la red</i>	342

<i>Funciones de activación</i>	343
<i>Funciones de activación sigmoidales</i>	343
<i>Unidades lineales rectificadas (ReLU)</i>	349
<i>Otras funciones de activación alternativas</i>	352
<i>Modos de entrenamiento</i>	354
<i>Entrenamiento por lotes</i>	355
<i>Entrenamiento online</i>	356
<i>Entrenamiento por minilotes</i>	360
<i>El conjunto de entrenamiento</i>	363
<i>Preprocesamiento de los datos de entrenamiento</i>	363
<i>Ampliación del conjunto de entrenamiento</i>	367
<i>Entrenamiento con adversario</i>	369
<i>Estudiantes y profesores</i>	375
<i>Inicialización de la red</i>	378
<i>Inicialización de los pesos de la red</i>	379
<i>Inicialización de los sesgos de las neuronas</i>	385
<i>Pre-entrenamiento de la red</i>	386
<i>Tasas de aprendizaje</i>	393
<i>Tasas locales de aprendizaje</i>	400
<i>Momentos</i>	404
<i>Otras formas de actualizar los parámetros de la red</i>	407
<i>Normalización por lotes</i>	409
<i>Hiperparámetros</i>	412
<i>Ajuste manual</i>	412
<i>Ajuste automático</i>	415
<i>Neuroevolución</i>	424
<i>Prevención del sobreaprendizaje</i>	427
<i>El problema del sobreaprendizaje</i>	428
<i>Regularización</i>	432
<i>Regularización de la función de coste</i>	434
<i>Restricciones sobre los parámetros de la red</i>	445

<i>Introducción de ruido</i>	450
<i>Introducción de ruido sobre las entradas</i>	451
<i>Introducción de ruido sobre los pesos</i>	453
<i>Introducción de ruido sobre las capas ocultas</i>	454
<i>Early stopping</i>	455
<i>Ensembles</i>	459
<i>Dropout</i>	462
<i>Algoritmos de optimización</i>	471
<i>El caso unidimensional</i>	474
<i>Búsqueda ternaria</i>	475
<i>Búsqueda de la sección áurea</i>	476
<i>Interpolación parabólica inversa</i>	477
<i>El método de Brent</i>	478
<i>¿Podemos usar el valor de la derivada de la función?</i>	479
<i>El caso multidimensional</i>	479
<i>Mínimos locales y otros puntos críticos</i>	483
<i>Gradiente descendente estocástico</i>	487
<i>Promediado de Polyak</i>	490
<i>Momentos</i>	492
<i>Momentos tradicionales</i>	494
<i>Momentos de Nesterov</i>	496
<i>Tasas de aprendizaje adaptativas</i>	497
<i>AdaGrad</i>	500
<i>AdaDelta</i>	501
<i>Rprop</i>	503
<i>RMSprop</i>	506
<i>vSGD</i>	508
<i>AdaSecant</i>	509
<i>Adam</i>	511

<i>Técnicas de optimización de segundo orden</i>	514
<i>El método de Newton</i>	520
<i>El método de Gauss-Newton</i>	522
<i>El método de Levenberg–Marquardt</i>	524
<i>Métodos quasi-Newton</i>	527
<i>Variantes de BFGS</i>	530
<i>Gradientes conjugados</i>	534
<i>Aprendiendo a optimizar de forma automática</i>	538
<i>Paralelización de las técnicas de optimización</i>	540
<i>Implementación asíncrona paralela del gradiente descendente</i>	542
<i>Implementación asíncrona distribuida del gradiente descendente</i>	543
<i>Paralelización de otras técnicas de optimización</i>	547
<i>Comentarios finales</i>	548
<i>PARTE III. ARQUITECTURAS ESPECIALIZADAS</i>	553
<i>Softmax</i>	555
<i>La entropía cruzada como función de coste</i>	558
<i>Curso rápido de Teoría de la Información</i>	559
<i>Entropía</i>	560
<i>Entropía condicionada e información mutua</i>	564
<i>Entropía relativa: Divergencia Kullback-Leibler</i>	565
<i>Entropía cruzada</i>	566
<i>La función softmax</i>	569
<i>Estimación de máxima verosimilitud</i>	575
<i>Para problemas de regresión</i>	576
<i>Para otros tipos de problemas</i>	577
<i>Una derivación alternativa</i>	578
<i>Cuestiones de implementación</i>	581
<i>Estabilidad numérica</i>	581
<i>La función de coste</i>	583

<i>El gradiente de la función de coste</i>	586
<i>La capa softmax</i>	587
<i>Inicialización de los pesos</i>	589
<i>Técnicas de regularización</i>	590
<i>Aplicaciones</i>	592
<i>Mecanismos de atención</i>	594
<i>Apéndice: El gradiente natural</i>	596
<i>Redes convolutivas</i>	599
<i>Orígenes: El Neocognitrón de Fukushima</i>	601
<i>La operación de convolución</i>	606
<i>Capas convolutivas</i>	614
<i>Hiperparámetros de una capa convolutiva</i>	615
<i>Capa convolutiva vs. capa completamente conectada</i>	619
<i>Implementación</i>	621
<i>Implementación para GPU</i>	625
<i>Variantes</i>	626
<i>Capas de pooling</i>	627
<i>Hiperparámetros de una capa de pooling</i>	630
<i>Implementación</i>	631
<i>Implementación para GPU</i>	633
<i>Críticas</i>	634
<i>Cuestiones prácticas</i>	636
<i>Arquitectura de las redes convolutivas</i>	637
<i>Entrenamiento de redes convolutivas</i>	642
<i>Aprendizaje por transferencia</i>	645
<i>Visualización de las redes convolutivas</i>	648
<i>Seguridad de las redes convolutivas</i>	650
<i>Aplicaciones</i>	651
<i>Clasificación de imágenes</i>	653
<i>Detección de objetos</i>	669
<i>Reconocimiento facial</i>	673
<i>Procesamiento de imágenes</i>	676
<i>Síntesis de imágenes: Arte neuronal</i>	679

Bibliografía 687

Índice alfabético 765

*A los que aún conservan
su curiosidad intacta...*

Prefacio

Las redes neuronales artificiales, más conocidas actualmente bajo la denominación anglosajona de *deep learning*, están de moda. En realidad, es la tercera vez en la historia de la Inteligencia Artificial en la que algo así sucede. Tras experimentar varios ciclos expansivos y sufrir severas correcciones que las pusieron al borde de la desaparición, actualmente disfrutan de un auge espectacular que, en cierto modo, se asemeja al experimentado por los sistemas expertos en los años 80 o la minería de datos en el cambio de siglo. El boom de la industria de los sistemas expertos ya pasó a la historia. Sin embargo, la minería de datos ha ido evolucionando hasta transformarse en “ciencia de datos”, el trabajo más sexy del siglo XXI según Hal Varian, economista jefe de Google. Del mismo modo, todo parece indicar que los grandes logros de las técnicas basadas en *deep learning* a la hora de resolver problemas que se habían mostrado intratables hasta la fecha harán gozar a las redes neuronales artificiales de una nueva época dorada. Según la consultora Gartner, el 80 % de los científicos de datos usarán herramientas de *deep learning*.

Aunque los orígenes de las redes neuronales artificiales se remontan a los comienzos de la Inteligencia Artificial, en los años 40 del siglo XX, y el *deep learning* en sí despegaría en 2006, el éxito real del *deep learning* se debe a su aplicación práctica en problemas de interés para la industria tecnológica, como el reconocimiento de voz o la visión artificial. Su victoria rotunda en competiciones como ImageNet en 2012 puso de nuevo las redes neuronales artificiales en el radar de muchas empresas.

La demanda de expertos en *deep learning* ha crecido exponencialmente en unos pocos años, de casi no existir en 2014 a decenas de miles de ofertas de empleo. Parte de esa demanda proviene de gigantes tecnológicos como Google, Microsoft, Amazon, Apple, Facebook, Intel, NVidia, Netflix o Baidu. Esas empresas contratan a gran cantidad de ingenieros, además de adquirir pequeñas compañías especializadas en *deep learning*, con el objetivo de complementar su oferta de productos y servicios (además de protegerse frente a los avances de su competencia).

Pero no sólo las empresas tecnológicas requieren personal especializado en *deep learning*. En un mundo global en el que las empresas gozan de ventajas competitivas al ser de las primeras en adoptar nuevas tecnologías,

multitud de empresas de todos los sectores económicos intentan captar el talento necesario para desarrollar sistemas que le saquen partido a las herramientas que nos ofrece la Inteligencia Artificial y, más en particular, el *deep learning*.

Hay quien prefiere ver estas herramientas como cajas negras que se pueden incorporar, como un módulo más, a un proyecto ya en marcha. Es la estrategia de los que se limitan a utilizar soluciones suministradas por terceros. En el caso del *deep learning*, estaríamos hablando de herramientas como Keras, TensorFlow, MXNet, Microsoft Cognitive Toolkit, Apache SINGA, Torch, Caffe o el ya difunto Theano. Sí, pese a su juventud, algunas herramientas de *deep learning* ya son historia.

Sin embargo, como indica James Gautrey, gestor de Schroders, cuando todas las soluciones son adquiridas a proveedores externos, la velocidad de adopción se convierte en el único factor diferencial. Los que quieran disfrutar de un crecimiento sostenido a más largo plazo, no obstante, tendrán que ser capaces de desarrollar su propia tecnología. El libro que tiene en sus manos omite deliberadamente el uso de herramientas particulares por dos motivos. En primer lugar, las herramientas de moda cambian y evolucionan a gran velocidad, por lo que el libro quedaría obsoleto en muy poco tiempo (menos mal que no opté por Theano ;-). En segundo lugar, no resulta difícil encontrar múltiples tutoriales en Internet en los que se describe detalladamente el uso de herramientas particulares.

Lo que es mucho más difícil de conseguir es una formación sólida que siente las bases de un área de conocimiento, al margen de las modas del momento y de las peculiaridades de las herramientas disponibles. Por esta razón, este libro pretende ofrecer una perspectiva general del *deep learning* como disciplina dentro de la Inteligencia Artificial, sin olvidar sus orígenes ni sus avances más recientes (y tampoco sus limitaciones, que también las tiene). En lugar de un recetario para aplicar redes neuronales en problemas particulares usando la biblioteca X o un manual del tipo “aprenda redes neuronales sin esfuerzo”, aquí encontrará los principios fundamentales de las redes neuronales artificiales y la filosofía que dio lugar al *deep learning*. Una vez que sea capaz de comprender sus bases, le resultará sencillo entender nuevas ideas que se vayan proponiendo en el futuro y podrá adaptarse sin problemas a las idiosincrasias de las herramientas que se usen en la práctica cuando Keras, TensorFlow y compañía hayan pasado a la historia.

Tal como comenta Richard Thaler, premio Nobel de Economía, de los muchos aspectos únicos que ofrece trabajar de profesor en una universidad, el más valioso es disponer de la libertad de poder dedicarle tiempo a pensar en cualquier cosa que a uno le parezca interesante y seguir llamándolo trabajo. También ayuda coincidir con las personas adecuadas. Aquí he de agradecer a Miguel Delgado, de la Universidad de Granada, la confianza

que mostró en mí desde el primer momento y la libertad que siempre me dio para organizarme de la forma que creyese más conveniente en las distintas asignaturas que he tenido la oportunidad de compartir con él. De hecho, Miguel fue el que me propuso el reto de explicarles redes neuronales y *deep learning* a los alumnos de Inteligencia Computacional en el máster profesional en Ingeniería Informática de la Universidad de Granada. A esos alumnos les tocó sufrir las distintas versiones de unos apuntes que han ido evolucionando hasta dar lugar a este libro. Uno de ellos, Gustavo Rivas, ha contribuido con sus minuciosos comentarios y sugerencias a pulir numerosas aristas del texto.

Un trabajo de la envergadura de redactar un libro como éste tampoco sería posible si uno no se encontrase con personas que muestran un interés genuino en aprender ideas nuevas y, con su esfuerzo continuado, el deseo de hacer las cosas bien, sin buscar atajos fáciles ni anteponer fines a medios. En particular, mi agradecimiento especial a Alejandro Avilés, que compaginó su TFM sobre forma y contenido en redes neuronales con su trabajo en el CERN en Suiza, y a Víctor Martínez, con el que he tenido el privilegio de trabajar estos últimos años.

Y, por supuesto, tampoco puedo olvidarme de Juan Carlos Cubero, con el que llevo ya muchos años colaborando, ni de Ignacio Requena, responsable de mi primer contacto serio con las redes neuronales artificiales, en una asignatura de Neurocomputación que cursé allá por el siglo pasado, literalmente (fue en 1999).

A parte de mis compañeros en el Departamento de Ciencias de la Computación e Inteligencia Artificial de la UGR (Miguel Delgado, Víctor Martínez, Juan Carlos Cubero e Ignacio Requena), también han revisado borradores de los distintos capítulos de este libro otras personas que han contribuido a este proyecto dedicándole parte de su tiempo: José Luis Polo y Aída Jiménez, antiguos doctorandos del grupo de investigación IDBIS; Gustavo Rivas y Julio Omar Palacio, actuales doctorandos de IDBIS; Javier Portilla, del Instituto de Óptica del CSIC en Madrid; Roberto José del Río y Alejandro Moreno, del máster en ciberseguridad de la UGR; Ismael González y Ángel de Jaén, del máster en ciencia de datos de la UGR; Luis Castro, Luis Balderas, Nuria Rodríguez, Carlos Paradela e Ignacio Aguilera, del doble grado en Ingeniería Informática y Matemáticas de la UGR. Sin olvidar a Jesús Alberto Garrido, Rubén Escabia o César Aguilera, con los que tuve el honor de coincidir cuando ellos eran estudiantes de Ingeniería Informática en la UGR. Con sus comentarios y sugerencias, todos ellos han contribuido a hacer este libro un poco mejor.

Si la versión actual de este libro no recoge todas sus sugerencias de mejora es culpa mía, de mi pereza a la hora de volver a darle otra vuelta más a la organización del texto. Daniel Kahneman, colaborador de Richard Thaler, veía la pereza como algo positivo en Thaler, algo que le

permitía centrarse en las cuestiones realmente interesantes, aquéllas que superaban su tendencia natural a evitar el trabajo. En el caso de Thaler, la clave estaba en evitar la procrastinación, la postergación de actividades realmente importantes por otras más irrelevantes o agradables. En el caso de este libro, también influye la necesidad de cumplir con los plazos límite que me había autoimpuesto, para lo que no queda más remedio que establecer prioridades. Tal como ya descubriese el matemático Selmer Johnson en 1954, al analizar el primer algoritmo de planificación óptima: se empieza por las tareas que se completan más rápidamente... hasta que se acaba el tiempo disponible. No da tiempo a completar todo lo que uno querría, pero al menos se crea una (falsa) sensación de progreso, aunque sea a costa de una inversión de prioridades que se asemeja peligrosamente a la procrastinación. Curiosamente, Johnson describió su algoritmo de planificación en el contexto de la publicación de libros, en los que una primera máquina los imprime y una segunda máquina los encuaderna.

En realidad, un libro nunca se termina realmente, sino que uno se da por vencido y lo termina abandonando. En cuanto al resultado final, que espero que sea del agrado del lector, encontrará algunas peculiaridades no demasiado frecuentes:

- Los números se representan en notación anglosajona, con el punto reservado para los decimales. Es algo habitual para cualquier informático acostumbrado a los lenguajes de programación actuales. La coma, a la hora de representar cifras, se utiliza muy esporádicamente para separar conjuntos de dígitos en cifras numéricas muy elevadas, en ocasiones por encima del googol, 10^{100} , y sólo para mejorar su legibilidad.
- Los anglicismos son muy habituales en el vocabulario técnico, tal vez demasiado. En ocasiones, no existe en castellano una palabra equivalente con el mismo significado. Más a menudo, los términos en inglés no siempre se traducen igual al español, por lo cual el uso de un término u otro podría dar lugar a confusiones. De hecho, es habitual que algunos términos ni siquiera se lleguen a traducir (como es el caso de *deep learning*). Por este motivo, siempre que se utiliza un término en sentido técnico, se especifica el término inglés entre corchetes, notación que se emplea, además, para indicar el significado de los acrónimos que plagan el lenguaje técnico-científico.
- Aunque se ha intentado que el libro resulte más o menos autocontenido para personas que tengan cierta formación científica o técnica (entre los que se encontrarían informáticos, matemáticos, físicos, ingenieros, estadísticos o científicos de datos), puede que aquí o allá se encuentre con más de un término que le resulte desconocido. En tal caso, en sólo unos milisegundos puede encontrar una descripción del término en

cuestión mucho más acertada que la que yo podría proporcionarle en unas líneas. Sólo tiene que realizar una consulta rápida en su buscador favorito de Internet o, directamente, en Wikipedia.

- Los algoritmos descritos en este libro aparecen especificados en pseudocódigo. Este pseudocódigo debería ser fácilmente interpretable por cualquier persona que haya programado alguna vez utilizando un lenguaje de tipo imperativo. No es código Java, ni C/C++, ni Python, ni MATLAB... aunque toma prestadas características de todos estos lenguajes de programación. En particular, se utilizan espacios (como en Python) en lugar de llaves (como en C/C++ o Java) para delimitar bloques de código, como simple medida de economía de espacio.

En principio, no debería resultarle difícil trasladar el pseudocódigo al lenguaje de programación preferido por cada lector. Sólo ha de tener cuidado a la hora de manejar los índices de los *arrays*, con base 1 en lenguajes como MATLAB, con base 0 en lenguajes derivados de C.

Podría haber escogido alguno de los lenguajes ya existentes, ya que cualquiera de los citados goza de excelente salud en la actualidad, aunque he optado por molestar de forma uniforme a los defensores a ultranza de cada uno de esos lenguajes.

- El libro, como tendrá oportunidad de comprobar, hace caso omiso del consejo al que hace referencia Stephen Hawking en su *Breve historia del tiempo*. Según este consejo editorial, por cada ecuación que se incluye en un libro, sus ventas se reducen a la mitad. Hawking decidió no poner ninguna en su superventas, aunque al final sí que incluyó la famosa ecuación de Einstein ($E = mc^2$) con la esperanza de que sus lectores no se redujesen a la mitad. En este libro, dado su carácter técnico, las ecuaciones abundan. De hecho, hasta se encontrará de refilón con las ecuaciones de Maxwell. Espero, no obstante, que eso no haga que su número de lectores potenciales tienda asintóticamente a cero. En mi editorial no estarán demasiado contentos con ello.
- Este libro ha sido maquetado en L^AT_EX utilizando una plantilla basada en el estilo empleado por Edward R. Tufte en sus publicaciones¹. Dicha plantilla está disponible en <https://tufte-latex.github.io/tufte-latex/> y se puede utilizar sin restricciones gracias a su licencia Apache.

Personalmente, escribir este libro me ha servido para poner en orden multitud de ideas con las que me he ido familiarizando durante años. Como se suele decir, no se comprende realmente algo hasta que uno no es capaz de explicárselo claramente a otro (aunque la claridad del texto tendrá que juzgarla el lector por sí mismo).

Conforme se vayan desgranando ideas, puede que el lector se encuentre con algunas divagaciones, más o menos acertadas, que tienen como

¹ Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, 2001. ISBN 0961392142

objetivo establecer asociaciones de ideas potencialmente interesantes (a veces, sólo para el autor ;-). En cierto modo, se trata de fomentar la serendipia, término acuñado por Horace Walpole en 1754 a partir del cuento de los tres príncipes de Serendipia (el nombre arcaico de Sri Lanka). De acuerdo a la historia, los príncipes siempre hacían descubrimientos, por accidente, de cosas que no estaban buscando realmente. Al fin y al cabo, como se suele decir, la frase más emocionante para un científico, la que anticipa nuevos descubrimientos, no es “¡Eureka!” sino “es curioso”.

En cuanto al estilo del texto, se ha intentado que el resultado resulte legible para todo aquél que tenga un interés genuino en aprender acerca de redes neuronales artificiales y *deep learning*. Su relación señal-ruido [SNR: *Signal-to-Noise Ratio*] es, por tanto, inferior a la habitual en tratados académicos al uso. Se pretende, de esta forma, que su lectura resulte menos árida para el profano, aun a costa de introducir cierta redundancia en el texto, especialmente si lo comparamos con otras monografías “más matemáticas” sobre el tema. También se ha intentado que los distintos capítulos sean, en cierto modo, autocontenidos, para evitar que el lector tenga que acordarse absolutamente de todo lo que se dijo anteriormente para llegar a comprender detalladamente una sección concreta. Esta decisión, que se podría intentar justificar desde un punto de vista pedagógico, puede resultar algo molesta a ciertos lectores, ya que introduce una nueva fuente de ruido y un mayor grado de redundancia a nivel global. Por supuesto, esto no quiere decir que la relación señal-ruido de esta monografía llegue a los extremos de los libros que uno solía encontrar en los aeropuertos cuando iba de viaje, en la época anterior al *e-book* que a algunos lectores les resultará ya lejana, cuando no desconocida. Comprender los fundamentos de una disciplina técnica conlleva su esfuerzo y dicho esfuerzo no se puede eliminar por completo. Me conformo con que, una vez realizado ese esfuerzo, la experiencia le resulte gratificante y termine con la convicción de que mereció la pena.

A la hora de enfrentarse al desafío de dominar un nuevo conjunto de técnicas, muchos libros suelen incluir ejercicios y problemas al final de cada capítulo. En este caso, no los encontrará. Los ejercicios y problemas se diseñan, habitualmente, para afianzar conceptos, si bien sólo resultan verdaderamente útiles para los lectores de un perfil determinado (p.ej. estudiantes de una asignatura evaluada resolviendo problemas similares en un examen). Sin embargo, entre los lectores potenciales de este libro, se encuentra un público bastante heterogéneo, en el que me gustaría encontrar desde estudiantes que han oído hablar del tema y quieren ponerse al día, científicos con problemas específicos para los que la IA tiene aportaciones que realizar, o ingenieros encargados de desarrollar sistemas basados en redes neuronales artificiales; hasta gestores de proyectos que quieren incorporar técnicas de *deep learning* en sus productos, economistas que analizan tendencias o, simplemente, personas cuya cu-

riosidad topó con este libro por mero azar (la serendipia haciendo de las suyas). En ausencia de series encorsetadas de ejercicios y problemas, mi recomendación es que intente aplicar lo que aquí vaya aprendiendo en un problema que realmente le importe. Encuentre un proyecto en el que se puedan aplicar las técnicas aquí analizadas y hágalo suyo. Eso le permitirá aprender mucho más de lo que puede aprender limitándose a leer un libro, aunque el libro incluya multitud de cabos sueltos para que el lector los ate.

El libro va acompañado de una página web, <http://deep-learning.ikor.org>, en la que encontrará enlaces de interés y material complementario. Cualquier comentario, sugerencia o crítica será siempre bienvenida. No dude en ponerse en contacto conmigo para compartir sus impresiones y dudas sobre el mundillo de las redes neuronales artificiales y el *deep learning*.

Fernando Berzal
berzal@acm.org

La Zubia, Granada, junio de 2017 - mayo de 2018.

PARTE I. REDES NEURONALES ARTIFICIALES

En la que se sitúa en su contexto la Inteligencia Artificial, el aprendizaje automático y las redes neuronales, más conocidas actualmente por su denominación de moda: aprendizaje profundo o deep learning.

Inteligencia Artificial

Demasiada gente parece preocupada por la futura llegada de máquinas inteligentes. Hay quien incluso propone tomar medidas para evitarla, incluyendo la prohibición de determinados desarrollos, algo que históricamente se ha demostrado que no suele funcionar. Algunos parecen conformarse con aminorar, en cierta medida, el impacto económico y social de la llegada de la I.A., mediante la creación de impuestos especiales (la medida favorita de los políticos de todo pelaje). Sin embargo, tanto unos como otros ignoran que la I.A. ya está aquí y nos rodea en tareas tan cotidianas como realizar búsquedas en Internet, recibir recomendaciones personalizadas de todo tipo, leer nuestros mensajes de correo o movernos de un sitio a otro con ayuda de un navegador GPS. En este capítulo, repasamos la evolución de la I.A. desde sus orígenes, algunas de sus aplicaciones más conocidas y las características que todas ellas comparten.

¿Es usted de los que teme que las máquinas puedan llegar a desarrollar tal nivel de inteligencia que le dejen sin trabajo? Si tiene este libro en sus manos, puede estar relativamente tranquilo, al menos por el momento. Es más que probable que usted pertenezca a lo que los economistas denominan el sector cuaternario: la parte del sector servicios que incluye aquellos servicios que no se pueden automatizar por completo. A día de hoy, resulta imprescindible la presencia humana en investigación, en sanidad, en educación, en la transmisión de conocimientos o en cualquier otra actividad relacionada con el valor intangible de la información, como la informática, la consultoría o el diseño.

Si usted trabaja en el sector industrial, habrá comprobado cómo la mecanización permite aumentar espectacularmente la productividad de una fábrica reemplazando operadores humanos por máquinas y robots cada día más eficientes y versátiles. Ese aumento de productividad no se ha producido en el sector cuaternario: la productividad de un médico o de un profesor no ha cambiado tan radicalmente (aunque sí puedan ofrecer un mejor servicio gracias a determinados avances tecnológicos). Esa divergencia en las mejoras productivas de diferentes sectores económicos conduce a lo que el economista William Baumol denominaba ‘enfermedad de costes’: el aumento relativo de los costes asociados a tareas que no experimentan un aumento de la productividad laboral frente a aquéllas que sí lo experimentan.

Si usted trabaja en el sector servicios, pero en una actividad no relacionada directamente con el conocimiento, entonces sí debería estar algo inquieto. Digamos, por ejemplo, que usted es taxista. En vez de

centrar sus esfuerzos en pretender impedir la irrupción legal de compañías como Uber, Lyft, Hailo o Cabify, que facilitan la contratación de un vehículo con conductor a través de una aplicación móvil, debería estar más que preocupado por la llegada de los vehículos autónomos, que prescinden por completo del conductor. Vehículos, por cierto, que utilizan múltiples técnicas de Inteligencia Artificial para planificar sus rutas y responder frente a imprevistos en la carretera. Redes neuronales incluidas.

¿De dónde provienen los temores, a menudo infundados, sobre el desarrollo de la Inteligencia Artificial? En parte, de los mitos que ocupan en el mundo actual el rol de la mitología clásica en Grecia y Roma: las superproducciones de Hollywood.

- En *2001: Una odisea en el espacio* [2001: A Space Odyssey], película dirigida por Stanley Kubrick en 1968, un personaje clave es el ordenador de a bordo de una nave espacial con destino a Júpiter, denominado HAL 9000 en referencia a IBM, la mayor compañía tecnológica del momento. Ante la intención de los astronautas de desconectar la máquina, HAL comienza una lucha por su propia supervivencia, con consecuencias fatales para los astronautas.
- En *Colossus: El proyecto prohibido* [Colossus: The Forbin Project, Joseph Sargent, 1970], ambientado en plena Guerra Fría entre Estados Unidos y la U.R.S.S., el ordenador Colossus y su homólogo soviético, Guardian, toman el control del arsenal nuclear de las dos grandes superpotencias, entonces las únicas con capacidad nuclear, y acaban dominando el mundo (con todas las peculiaridades de la época, algo chocantes hoy en día).
- En *La fuga de Logan* [Logan's Run, Michael Anderson, 1976], una distopía futurista en la que los humanos del siglo XXIII disfrutan de una sociedad aparentemente idílica en la que no es necesario trabajar, ya que los servomecanismos se encargan de todo, un ordenador omnipoente impide que los humanos vivan más de 30 años para mantener la superpoblación a raya.
- En *Engendro mecánico* [Demon Seed, Donald Cammell, 1977], basado en una novela de Dean R. Koontz, un científico crea un supercomputador orgánico llamado Proteus que toma el control de los sistemas domóticos de su casa y construye pseudópodos de aleaciones amorfas con el objeto de tener descendencia... con la mujer del científico.
- En *La guerra de las galaxias* [Star Wars, George Lucas, 1977], el androide traductor C-3PO y el robot lleno de recursos R2-D2 (“Arturito”) acompañan fielmente en sus andanzas a los personajes humanos de la saga.

AVISO: Sáltese la siguiente lista si no quiere que le estropee el final de alguna de las películas que aparecen a continuación, protagonizadas todas ellas por alguna forma de Inteligencia Artificial. Perdón por los *spoilers* pero, parafraseando a Tolstoi en *Anna Karenina*, aunque muchas puedan parecer similares, cada una de ellas da una visión errónea de la Inteligencia Artificial a su manera...

- En *Alien, el octavo pasajero* [Ridley Scott, 1979], además del xenomorfo que protagoniza la película, otro de los pasajeros a bordo del Nostromo resulta ser un androide con agenda propia (e incentivos más alineados con los del alien que con los del resto de la tripulación).
- En *Blade Runner* [Ridley Scott, 1982], otra distopía futurista situada en la ciudad de Los Ángeles en 2019, a un ex-policía encarnado por Harrison Ford se le encarga la misión de perseguir y eliminar 'replicantes', androides de aspecto humano que han llegado a la Tierra ilegalmente procedentes de colonias en el espacio donde desempeñaban el trabajo para el que fueron inicialmente diseñados.
- En *TRON* [Steven Lisberger, 1982], un hacker interpretado por Jeff Bridges es atrapado por un ordenador y se ve obligado a participar en una especie de juego de gladiadores orquestado por un dictatorial MCP [*Master Control Program*], algo así como un sistema operativo dotado de inteligencia artificial. Al tratarse de una película de Disney, no se dibuja un panorama tan sombrío como en otras producciones: el hacker es apoyado por TRON, un programa de monitorización, con el que vencerá al malvado MCP.
- En *Juegos de guerra* [WarGames, John Badham, 1983], un hacker adolescente se cuela vía modem en el ordenador de alto secreto W.O.P.R. [*War Operations Plan Response*], apodado Joshua, que fue diseñado para tomar decisiones en caso de guerra nuclear con la Unión Soviética y al que hace creer que un ataque soviético está realmente sucediendo.
- En *Terminator* [James Cameron, 1984], un cyborg humanoide denominado Terminator y encarnado por Arnold Schwarzenegger es enviado desde el futuro para asesinar a la madre del futuro líder de la guerra de la humanidad contra las máquinas, que en el futuro dominarán el planeta tras una guerra nuclear iniciada por un ordenador denominado Skynet, encargado originalmente de controlar todos los sistemas de defensa de EE.UU..
- En *D.A.R.Y.L.* [Simon Wincer, 1985], el cyborg resulta ser un niño, que es adoptado por una familia que lo encuentra deambulando sin saber que D.A.R.Y.L. [Data Analyzing Robot Youth Lifeform] es el prototipo de un proyecto de investigación militar y que para los militares no es ya más que una pieza cara de hardware que hay que destruir.
- En *Cortocircuito* [Short Circuit, John Badham, 1986], el robot experimental Número 5, más conocido como Johnny 5, es electrocutado y eso le dota de inteligencia. Curiosa forma de lograrlo...
- En *RoboCop* [Paul Verhoeven, 1987], los robots ejercen de policías en Detroit, como un robot ED-209 [Enforcement Droid Series 209] al que

En realidad, los 'replicantes' muestran ser, en algunos aspectos, más humanos que los propios seres humanos, como sucede con Rachael (Sean Young) y también con Roy Batty (Rutger Hauer). Rachael ni siquiera es consciente de ser una replicante y son inolvidables las palabras finales de Roy. A pesar del cuidado ambiente futurista de la película, Harrison Ford llama por teléfono a Rachael... desde un videoteléfono de pago. Algo casi tan desconocido para muchos jóvenes actuales, acostumbrados a sus *smartphones*, como los cassettes, los vídeos Beta o los discos flexibles.

tendrá que hacer frente RoboCop, un cyborg que combina la mente de un oficial asesinado en acto de servicio y un cuerpo biomecánico.

- En algunas películas de la saga *Star Trek* de la década de los 90 aparece el comandante Data, un androide que ejerce de oficial de operaciones de la nave Enterprise (aunque muchos pensarán que el androide de la saga es el vulcaniano Spock, personaje interpretado por Leonard Nimoy desde 1966).
- En *Matrix* [hermanos Wachowski, 1999], un hacker llamado Neo (Keanu Reeves) se cuestiona la realidad y descubre que la verdad va mucho más allá de lo que podía imaginar. Una distopía más en la que las máquinas mantienen presos a los humanos para aprovechar su energía y les hacen creer que viven en una realidad virtual, Matrix. Esta realidad virtual es supervisada por agentes, programas que eliminan cualquier tipo de resistencia en el sistema, como el agente Smith.
- En *Inteligencia Artificial* [A.I., Steven Spielberg, 2001], un niño artificial llamado David, capaz de amar, es utilizado para reemplazar a un niño enfermo mantenido en criostasis hasta que se descubra una cura para su enfermedad. Tras la curación del niño, David es abandonado y acaba emprendiendo una huida con otro robot, un gigoló mecatrónico llamado Joe.
- En *Yo, robot* [I, Robot, Alex Proyas, 2004], película basada en un libro de Isaac Asimov, un supercomputador llamado VIKI [Virtual Interactive Kinetic Intelligence] que gestiona las operaciones de la compañía de robótica U.S. Robotics (USR), decide que las tres leyes de la robótica han de retocarse para proteger a los humanos de sí mismos.
- En *La conspiración del pánico* [Eagle Eye, D.J. Caruso, 2008], un supercomputador cuántico llamado ARIIA [*Autonomous Reconnaissance Intelligence Integration Analyst*] orquesta un golpe de Estado poniendo en marcha la “Operación Guillotina” tras decidir que el presidente del gobierno de Estados Unidos es una amenaza para el bien común.
- En la película de dibujos animados *WALL-E* [Andrew Stanton, 2008], un robot recolector de basura llamado WALL-E [*Waste Allocation Lift Loader, Earth-Class*] queda prendado de un robot de reconocimiento llamado EVE [*Extra-terrestrial Vegetation Evaluator*].
- En *Moon* [Duncan Jones, 2009], un astronauta trabaja en una explotación minera en la Luna con la única compañía de GERTY, un robot inteligente con la voz de Kevin Spacey, en lo que no parece ser más que un contrato temporal por 3 años... hasta que descubre la verdad tras esa explotación minera en el espacio.
- En *Her* [Spike Jonze, 2013] es Scarlett Johansson la que da voz a Samantha, el primer sistema operativo inteligente (“no sólo un sistema

Las tres leyes de Asimov establecen que:
 (1) un robot nunca hará daño a un ser humano o, por inacción, permitirá que un ser humano sufra daño; (2) un robot siempre ejecutará las órdenes dadas por los seres humanos, excepto si estas órdenes entran en conflicto con la primera ley; y, por último, (3) un robot protegerá su propia existencia en la medida en que dicha protección no entre en conflicto con las dos leyes anteriores.

operativo, una conciencia") del que un escritor solitario interpretado por Joaquin Phoenix acaba enamorándose. El sistema operativo va evolucionando en sus anhelos, desarrolla rasgos humanos como los celos y, finalmente, acaba abandonando a su embelesado usuario para irse con otros de su misma, digamos, especie.

- En *Ex Machina* [Alex Garland, 2014], Alicia Vikander da cuerpo a una humanoide llamada Ava, diseñada para superar el test de Turing y capaz de flirtear con humanos para manipularlos con tal de lograr sus propios objetivos.

Tras este somero repaso a algunas de las películas de ciencia ficción más populares de los últimos cincuenta años, no es de extrañar que el público en general pueda no ver la Inteligencia Artificial con buenos ojos. Salvando las películas destinadas a un público infantil, en conjunto se ofrece una perspectiva negativa de los avances tecnológicos que nos puede ofrecer el desarrollo de la I.A., entre otras cosas porque eso les suele dar mucho más juego a los guionistas de Hollywood. Si Mary Shelley hubiese vivido durante la segunda mitad del siglo XX, su *Frankenstein*, o *El Moderno Prometeo* habría sido muy posiblemente un robot. Si hubiese presenciado la explosión de Internet a comienzos del XXI, la creación del doctor Frankenstein viviría probablemente en la nube.

En realidad, la aversión inicial a cualquier novedad tecnológica que pueda afectar al statu quo no es algo precisamente nuevo. Ned Ludd, cuyo nombre real posiblemente fuese Edward Ludlam, destrozó un par de telares en plena Revolución Industrial, a finales del siglo XVIII. Ya en el siglo XIX, el sabotaje de telares se solía atribuir a los seguidores del autoproclamado capitán, general o incluso rey Ludd, de donde proviene el término luddita. Hoy en día, nos puede parecer ridícula la destrucción de telares bajo el pretexto de que destruyen empleo manual y nadie defiende que las prendas de vestir se sigan tejiendo manualmente, entre otras cosas porque el aumento de productividad asociado a los avances tecnológicos permite el acceso a productos que, hasta hace no demasiado tiempo, estaban sólo al alcance de los más ricos.

El economista francés Frédéric Bastiat supo verlo en el siglo XIX y publicó, como parte de sus *Sofismas económicos*, una conocida sátira en la que los fabricantes de velas realizan una petición a la Cámara de los Diputados para proteger su sector económico frente a la competencia desleal de una potencia extranjera: el Sol. Las supuestas ventajas económicas que incluía Bastiat en su sátira podrían, igualmente, aplicarse a la defensa de la iluminación mediante velas frente al auge de la bombilla eléctrica, al mantenimiento del coche de caballos frente a los vehículos con motor de combustión interna o a la conservación de los puestos de trabajo de amanuenses y tejedores frente al uso de la imprenta y el telar.

La mentalidad luddita pervive y reaparece frente a cada nuevo avance

Con cierta frecuencia, aparecen en los medios de comunicación manifiestos, cartas abiertas y predicciones agoreras sobre la autodestrucción del ser humano como consecuencia del desarrollo de la Inteligencia Artificial, en ocasiones firmadas por personalidades de renombre (que no siempre están al día del verdadero estado del arte, ya que normalmente no es la I.A. la especialidad por la que adquirieron su prestigio). El progreso, nos guste o no, no se detiene ante las proclamas ludditas. Aunque sí se puede ralentizar mediante trabas administrativas y nuevos impuestos, tal como propone Robert Schiller (premio Nobel de Economía por sus aportaciones a la economía financiera en el análisis de las ineficiencias del mercado, no por sus estudios sobre crecimiento económico). O, al menos, mitigar sus efectos con medidas como la renta básica universal, bajo el riesgo de que acabemos como los humanos de WALL-E...

tecnológico. Entre otros motivos porque el paro sí que puede aumentar temporalmente, mientras los trabajadores desplazados por la nueva tecnología encuentran una nueva ocupación (con mayores dificultades para los trabajadores de mayor edad) y se reorganizan los factores productivos de la economía. Los neoludditas, aunque más sutiles en sus formas habitualmente, siguen tan activos como siempre. Más aún, cuando ese avance tecnológico afecta de lleno a algo que, hasta ahora, creíamos exclusivo de los seres humanos: su inteligencia.

Uno de los temores más habituales en la actualidad es que los robots acaparen la producción de todos los bienes y la demanda no sea suficiente para absorber esa producción, especialmente si los humanos se han quedado en paro por ser menos productivos que los robots dotados de Inteligencia Artificial. Se ignora la principal aportación de otro economista francés, Jean-Baptiste Say, cuya ley de los mercados se puede resumir en que la oferta crea su propia demanda.² En caso de sobreproducción, la economía tenderá a retornar al equilibrio si los precios son flexibles y se permite que bajen. Esto puede llevar algo de tiempo, ya que la alarma creada por la deflación de precios puede ocasionar la acumulación de dinero. A la larga, sin embargo, no hay ningún límite a la demanda, tal como David Ricardo expuso en sus *Principios de Economía Política* de 1817.

En realidad, para todos aquellos que temen la llegada de la Inteligencia Artificial, una mala noticia: la Inteligencia Artificial ya está aquí.

Definiciones de I.A.

Ahora bien, para ser conscientes del alcance de la Inteligencia Artificial, antes deberíamos intentar delimitar su ámbito mediante su definición, para lo cual primero tendremos que determinar qué es lo que entendemos por inteligencia. Si cogemos un diccionario, no llegaremos demasiado lejos, ya que generalmente no nos encontraremos con más que una definición circular. Por ejemplo, el Diccionario de la Real Academia define inteligencia como:

1. Capacidad de entender o comprender.
2. Capacidad de resolver problemas.
3. Conocimiento, comprensión, acto de entender.

Si intentamos profundizar, nos dice que entender es conocer y que conocer es entender. Y, obviamente, que conocimiento es, además de acción y efecto de conocer, entendimiento, inteligencia o razón natural (sic). Nada que nos permita avanzar demasiado, más allá de lo de resolver problemas. En cuanto a la definición de ‘problema’, el diccionario no aporta mucho (“cuestión que se trata de aclarar”), si bien resulta interesante para un informático la definición de Jerry Weinberg, un conocido

² Pedro Schwartz. The fear of robots. *Econlib: Library of Economics and Liberty*, May 1, 2017. URL <http://www.econlib.org/library/Columns/y2017/Schwartzrobots.html?>

consultor: diferencia entre cómo se perciben y cómo se desean las cosas.³ Esto es, un problema es el espacio que hay entre lo que las cosas son y lo que nos gustaría que fuesen, lo que involucra necesariamente una perspectiva subjetiva. Dicho de otro modo, para resolver un problema, o bien cambiamos las cosas para que se adapten a nuestras preferencias, o bien somos capaces de actuar sobre nuestras preferencias para que se conformen con lo que realmente hay.

Si buscamos directamente la entrada correspondiente a inteligencia artificial en el Diccionario de la Real Academia obtenemos algo más de información: en Informática, “Disciplina científica que se ocupa de crear programas informáticos que ejecutan operaciones comparables a las que realiza la mente humana, como el aprendizaje o el razonamiento lógico”. Algo más cercano a una definición académica tradicional del campo de la Inteligencia Artificial.

A la hora de definir la I.A., algunos autores prefieren centrarse en sistemas que actúan, como los robots, mientras que otros prefieren focalizar su atención en el desarrollo de sistemas capaces de razonar o pensar. Tanto en un bando como en otro, hay quien define la I.A. con respecto al ser humano (asumiendo que es inteligente, algo de lo que uno puede tener sus dudas más que justificadas). Otros prefieren definir la inteligencia como un fenómeno asociado a la racionalidad, asumiendo que esa sea su característica definitoria.

Hay quien llega a proponer cambiarle el nombre a la Inteligencia Artificial para evitar sus problemas de imagen, que asocian la Inteligencia Artificial con distopías futuristas no demasiado halagüeñas o predicciones apocalípticas de los neoludditas. Jerry Kaplan, de Stanford, propone llamarla “computación antrópica” [*anthropic computing*], como término que engloba los esfuerzos dirigidos al diseño de sistemas informáticos de inspiración biológica, máquinas que imitan formas y capacidades humanas y sistemas que interactúan con personas de forma natural. Dan Klein y Pieter Abbeel, de Berkeley, sugieren “racionalidad computacional” [*computational rationality*] como nombre alternativo para la Inteligencia Artificial, haciendo hincapié en la maximización de la utilidad esperada del sistema como medida de su inteligencia (una perspectiva que quizá pekee de ser excesivamente economicista). En cualquier caso, esos términos alternativos carecen del glamour de la denominación tradicional y es poco probable que cuajen, por lo que seguiremos llamando a la Inteligencia Artificial por su nombre.

Dado que es prácticamente imposible alcanzar una definición de I.A. con la que todo el mundo esté de acuerdo, Keith Downing propone que nos conformemos con una definición mucho más ambigua y menos ambiciosa: hacer lo correcto en el momento adecuado, desde el punto de vista de un observador humano externo.⁴ Esta definición de caja negra se basa en el comportamiento observado del sistema y se centra más en el

³ Donald C. Gause y Gerald M. Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House, 1989. ISBN 0932633137

Recordemos que el propio Albert Einstein pensaba que sólo había dos cosas infinitas: el Universo y la estupidez humana. Y de la primera no estaba seguro.

⁴ Keith L. Downing. *Intelligence Emerging: Adaptivity and Search in Evolving Neural Systems*. MIT Press, 2015. ISBN 0262029138

resultado de un proceso (¿razonamiento? ¡tal vez! ¿pensamiento? ¡quia!) que en el proceso en sí, el cual de todas formas sólo comprendemos de forma muy rudimentaria.

Particularmente, me sigo quedando con la definición clásica de Elaine Rich, de la Universidad de Texas en Austin: el estudio de cómo hacer que los ordenadores hagan cosas que, por ahora, los humanos hacemos mejor.⁵ Ésta es una definición móvil de la I.A., ya que, en cuanto un problema abordado por técnicas de I.A. se resuelve satisfactoriamente, pasa a quedar automáticamente fuera de su ámbito de actuación. Por otro lado, mantiene perenne el interés de la I.A. centrando nuestra atención en aquellos problemas que, actualmente, los seres humanos somos capaces de resolver mejor que las máquinas.

Los problemas de la I.A.

Los orígenes de la I.A. se remontan a 1956. Durante el verano de ese año, un grupo de investigadores se dieron cita en el Darmouth College, una universidad privada de la Ivy League con sede en la ciudad de Hanover, New Hampshire, y que forma parte de las nueve instituciones coloniales de educación superior creadas antes de la independencia de Estados Unidos, declarada en 1776. Entre los asistentes a la cita se encontraban nombres muy conocidos de la talla de John McCarthy (creador del lenguaje de programación Lisp), Marvin Minsky (cofundador del laboratorio de I.A. del MIT), Nathaniel Rochester (diseñador del IBM 701 y autor del primer ensamblador simbólico), Claude Shannon (padre de la Teoría de la Información), Arthur Samuel (autor del primer programa de ordenador capaz de aprender), Herbert Simon (premio Nobel de Economía en 1978, además de premio Turing en 1975) o Allen Newell (que compartió el premio Turing de 1975 con Herbert Simon). A algunos de ellos nos los volveremos a encontrar más adelante. El caso es que, desde el nacimiento de la Inteligencia Artificial, un optimismo se apoderó de la veintena de investigadores que participaron en la reunión de Darmouth. De hecho, ya en la propuesta de organización del evento se partía de la siguiente conjectura inicial: “Cualquier aspecto del aprendizaje o de cualquier otra característica de la inteligencia puede describirse de manera tan precisa que se puede construir una máquina para simularlo”.

Obviamente, las cosas no resultaron tan fáciles como se preveía y aún hoy quedan muchos problemas por resolver. El objetivo final de la I.A. queda reflejado en un test propuesto por Alan Turing en 1950 que sirve para evaluar si un sistema artificial exhibe una conducta inteligente.⁶ Según el test de Turing, una máquina superaría el test si fuese capaz de lograr la eficiencia de un ser humano en todas las actividades de tipo cognitivo hasta el punto de ser capaz de engañar a un evaluador humano, el cual sería incapaz de determinar si estaba interactuando con

⁵ Elaine Rich. *Artificial Intelligence*. McGraw-Hill, 1983. ISBN 0070522618

⁶ Alan M. Turing. Computing machinery and intelligence. *Mind*, 59(236): 433–460, 1950. ISSN 00264423. DOI: 10.1093/mind/LIX.236.433. URL <http://www.jstor.org/stable/2251299>

una máquina o con otro ser humano.

Aunque todavía no se ha conseguido crear una máquina capaz de superar el test de Turing de forma general, los primeros éxitos en tareas particulares no tardaron demasiado en llegar. En los años 50, Allen Newell, Herbert Simon y Cliff Shaw trabajaban para RAND Corporation, un *think tank* que ofrecía servicios de I+D a las fuerzas armadas de Estados Unidos. Entre 1955 y 1956, escribieron el primer programa con Inteligencia Artificial, capaz de imitar las habilidades de resolución de problemas de un ser humano a la hora de demostrar teoremas: el Logic Theorist. Dicho programa fue capaz de demostrar 38 de los 52 primeros teoremas de los *Principia Mathematica* de Alfred Whitehead y Bertrand Russell. Posteriormente, en 1959, crearon un programa más sofisticado de demostración de teoremas denominado GPS [*General Problem Solver*.⁷ GPS incorporaba técnicas de búsqueda heurística y fue el primer programa que separó el conocimiento asociado a un problema (las reglas lógicas que representan un problema particular) de la estrategia que permite la resolución de problema.

Esos primeros éxitos dieron lugar a un optimismo desbordado. Por aquella época, Herbert Simon pensaba que las máquinas serían capaces de hacer cualquier tarea realizada por el hombre en apenas veinte años. Marvin Minsky acortaba el plazo hasta afirmar que en diez años se resolverían sustancialmente los problemas de inteligencia artificial. Por su parte, Claude Shannon se imaginaba, apostando por las máquinas, que nosotros seríamos para los robots como los perros para nosotros.

Pero no todo fueron éxitos y los resultados decepcionantes no tardaron demasiado en hacer su aparición, conduciendo a lo que parecían ser callejones sin salida. Por ejemplo, en plena Guerra Fría, había un gran interés en construir sistemas capaces de traducir automáticamente del ruso al inglés. La traducción automática [*machine translation*] resultó no ser tan fácil. Por ejemplo, si traducimos del inglés al ruso una frase como '*The spirit is willing but the flesh is weak*' (algo así como 'el alma está dispuesta pero la carne es débil') y luego traducimos el resultado del ruso al inglés, puede que obtengamos '*The vodka is good but the meat is rotten*' (el vodka es bueno pero la carne está podrida). Los traductores actuales, como los de Google, Microsoft o Facebook, siguen enfrentándose hoy en día a los innumerables matices del lenguaje natural, y no siempre con resultados mucho mejores que hace 50 años (haga la prueba si lo desea).

El informe ALPAC de 1966 causó la eliminación de la financiación gubernamental para sistemas de traducción automática en Estados Unidos. Un informe crítico similar, el informe Lighthill de 1974, afectó a la investigación en I.A. desarrollada en el Reino Unido. Si a eso unimos las críticas de Marvin Minsky y Seymour Papert a los modelos de redes neuronales artificiales de la época, con las que la investigación en redes

⁷ Allen Newell, John C. Shaw, y Herbert A. Simon. Report on a General Problem-Solving Program. En *Proceedings of the International Conference on Information Processing*, pages 256–264, 1959

neuronales casi desaparece, y la decepción en DARPA con los programas de reconocimiento de voz desarrollados en la Universidad de Carnegie-Mellon durante los años 70, no es de extrañar que la investigación en I.A. entrase en un período oscuro al que se denomina el invierno de la Inteligencia Artificial.

El interés por la Inteligencia Artificial no decayó por completo, sino que se reorientó a la resolución de problemas concretos de interés práctico. Por ejemplo, la búsqueda de caminos de coste mínimo aparece en diversas aplicaciones, entre ellas la planificación de trayectorias para vehículos autónomos. Peter Hart, Nils Nilsson y Bertram Raphael, del Stanford Research Institute en Menlo Park, California, idearon el famoso algoritmo A*. Dicho algoritmo es el epítome de las técnicas de búsqueda heurística. El algoritmo A* combina de forma elegante lo ya conocido g con una estimación heurística h de lo aún desconocido: $f = g + h$ es el criterio utilizado para guiar el proceso de búsqueda. Si la estimación heurística h es optimista, admisible según la terminología original, el algoritmo A* es óptimo en el sentido de que ningún otro algoritmo de búsqueda resolverá el problema en menos tiempo utilizando sólo la información proporcionada por la heurística.^{8,9}

Para muchos, no obstante, la clave de la Inteligencia Artificial no está en la resolución de problemas de búsqueda, sino en la representación formal del conocimiento. En los años 70 se diseñaron los primeros sistemas basados en el conocimiento: los sistemas expertos, así llamados porque pretendían complementar o sustituir las habilidades de expertos en dominios muy específicos que requerían un conocimiento experto no de fácil acceso. Entre los primeros sistemas expertos se encontraban sistemas para identificar compuestos orgánicos a partir de la información proporcionada por un espectrógrafo de masas (DENDRAL), para el diagnóstico y tratamiento de enfermedades infecciosas (MYCIN), para la exploración de posibles yacimientos minerales (PROSPECTOR) o para la elaboración de la lista de componentes necesarios para la configuración de un sistema informático (R1/XCON). En una época en la que un ordenador necesitaba una habitación para su instalación y la falta de un componente en el envío de un pedido podía suponer un coste significativo para la empresa, se estima que R1/XCON le supuso a Digital Equipment Corporation un ahorro de 25 millones de dólares anuales en la instalación de sistemas VAX, una arquitectura CISC de los años 70 que reemplazó a las máquinas PDP de DEC. El éxito de los primeros sistemas expertos dio lugar al nacimiento de la industria de la I.A. en los años 80, en la que se vivió un auténtico boom de empresas dedicadas al desarrollo de sistemas expertos. En cierto modo, dichos sistemas perviven hoy en día bajo el apelativo de motores de reglas [*business rule engines*], si bien dedicados a tareas más prosaicas, tales como la emisión de nóminas (pero no menos complejas, dada la voracidad legislativa y la maraña de normativas a las

⁸ Peter E. Hart, Nils J. Nilsson, y Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. ISSN 0536-1567. DOI: 10.1109/TSSC.1968.300136

⁹ Peter E. Hart, Nils J. Nilsson, y Bertram Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bulletin*, (37):28–29, 1972. ISSN 0163-5719. DOI: 10.1145/1056777.1056779

Digital, conocida en los años 90 por sus procesadores Alpha con arquitectura RISC de 64 bits, fue adquirida por Compaq en 1998. Posteriormente, Compaq se fusionó con Hewlett-Packard en 2002.

que están sujetas las empresas actuales).

Pese a no tener el impacto económico de otras áreas en las que se pueden aplicar las técnicas de Inteligencia Artificial, si existe un campo en el que la I.A. siempre ha despertado especial interés, es el de una de las capacidades humanas que nos diferencian de la mayor parte de los animales: el juego. Por ahora, centrémonos en lo que los economistas entienden por teoría de juegos;¹⁰ esto es, el estudio mediante modelos matemáticos de la interacción entre agentes racionales inteligentes. Los juegos de mesa siempre han servido como gancho publicitario para demostrar las capacidades supuestamente inteligentes de los ordenadores:

- El juego de las damas [*checkers* en EE.UU., *draughts* en Gran Bretaña] fue de los primeros para los que se desarrolló un programa de ordenador capaz de jugarlo. Arthur Samuel, de IBM, programó un ordenador para que aprendiese a jugar a las damas a finales de los años 50.¹¹ Tras algunas horas de práctica, el programa era capaz de jugar mejor que su programador. Internamente, utilizaba un algoritmo conocido como poda alfa-beta, que permite analizar las consecuencias de los movimientos de cada jugador sin explorar todas las ramas del árbol de búsqueda asociado al juego. En el momento en el que nos damos cuenta de que una estrategia no nos puede conducir a mejores resultados que otra que ya hemos analizado, no es necesario seguir explorando esa posibilidad.

De hecho, las damas fue el primer juego en el que un programa de ordenador fue capaz de ganar a un campeón mundial, Marion Tinsley. Fue un jugador artificial llamado Chinook en 1994. Al contrario del programa original de Samuel, no se puede decir que Chinook utilizase Inteligencia Artificial, ya que todo su conocimiento del juego provenía de sus programadores. Usaba una base de datos que definía el juego perfecto para todos los finales de partida con 8 o menos piezas (443748 millones de posiciones). Ya en 2007, la capacidad de cálculo de los ordenadores permitió explorar todas las posibilidades del juego y demostrar que, si no se cometiesen errores, todas las partidas de damas acabarían en tablas.¹²

Por tanto, desde el punto de vista de la I.A., las damas forman parte de la misma categoría de juegos que el infantil juego de Nim, las 3 en raya [*tic-tac-toe*] o el Conecta 4. El número de posiciones diferentes que se pueden dar en estos juegos son 362,280 (9! para las 3 en raya), 4,531,985,219,092 (del orden de 10^{13} para el Conecta 4) y 500,995,484,682,338,672,639 (más de 10^{20} para las damas). Números enormes, pero tratables con la capacidad de cálculo de los ordenadores actuales. Y, por tanto, muy aburridos de jugar frente a un ordenador, que resulta implacable y nunca pierde.

- El juego del ajedrez es mucho más complejo que el de las damas. Si

¹⁰ John von Neumann y Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 60th anniversary commemorative edition, 1947. ISBN 0691130612

¹¹ Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959. ISSN 0018-8646. DOI: 10.1147/rd.33.0210

¹² Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, y Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007. ISSN 0036-8075. DOI: 10.1126/science.1144079. URL <http://science.sciencemag.org/content/317/5844/1518>

en un turno determinado un jugador puede elegir entre b movimientos posibles y una partida puede durar d movimientos, el árbol asociado al juego tendrá del orden de b^d nodos (un árbol de profundidad d con factor de ramificación b). En el ajedrez, $b \approx 35$ y $d \approx 80$, por lo que no podríamos explorar el árbol completo del juego aunque dispusiésemos de todo el tiempo del mundo: 4×10^{17} segundos desde el Big Bang, insuficientes para explorar un árbol de 10^{123} nodos, número superior incluso al número de átomos del universo observable, que estará sobre 10^{80} . Aun así, el ordenador IBM Deep Blue¹³ fue capaz de derrotar al campeón mundial de ajedrez Gary Kasparov en mayo de 1997.

¿Cómo se consiguió esta hazaña? Básicamente, utilizando la fuerza bruta de un superordenador RS/6000 equipado con 30 procesadores P2SC de 120 MHz y 480 chips ASIC VLSI. El programa de ajedrez estaba programado en C y se ejecutaba bajo el sistema operativo AIX (la versión de UNIX para ordenadores IBM). Capaz de evaluar 200 millones de posiciones por segundo, estaba entonces en la lista de supercomputadores más potentes del mundo, con 11.38 GFLOPS. Para decidir su jugada, Deep Blue exploraba un árbol con una profundidad de entre 6 y 8 movimientos por jugador (a cada nivel del árbol se le denomina *ply*, por lo que exploraba árboles de 12 a 16 ply) si bien era capaz de llegar a explorar árboles de profundidad 40 en algunas situaciones (¡¡40 ply!!), lo que conseguía reduciendo el factor de ramificación del árbol desde 35 hasta 6 utilizando una poda alfa-beta. Para evaluar las hojas de ese árbol, que obviamente no solían corresponder a situaciones finales del juego, combinaba unas 8000 características y heurísticas diferentes.

¿Fue la fuerza bruta de un supercomputador la que venció realmente al campeón del mundo de ajedrez? No está del todo claro. En la sexta partida del enfrentamiento entre Deep Blue y Kasparov, Kasparov cometió un error en la apertura que acabó costándole la derrota por $3\frac{1}{2} - 2\frac{1}{2}$. Según cuentan, Kasparov había quedado perplejo por uno de los movimientos de Deep Blue en su primera partida. Sin entender el porqué de ese movimiento y pensando que debía existir una justificación lógica para un movimiento así, se pasó horas intentando buscarle una explicación y fue incapaz de descansar adecuadamente antes de las partidas posteriores. Por lo que se ve, puede que se tratase de un simple *bug* en el programa de ajedrez de Deep Blue... A diferencia de Kasparov, el juego de Deep Blue no se vio afectado por su estado de ánimo, por lo que acabó haciendo historia al vencer al campeón mundial de ajedrez. O tal vez fue Kasparov el que cayó derrotado por sus debilidades humanas.

A medio camino entre la complejidad de las damas y la del ajedrez se encuentra un juego llamado Reversi, comercializado por Mattel bajo

¹³ Murray Campbell, A. Joseph Hoane, y Feng-Hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1):57 – 83, 2002. ISSN 0004-3702. DOI: 10.1016/S0004-3702(01)00129-1. URL <http://www.sciencedirect.com/science/article/pii/S0004370201001291>

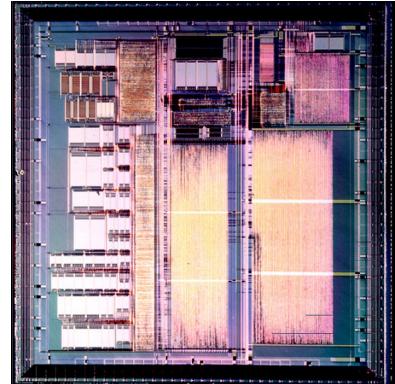


Figura 1: Chip ASIC [Application-Specific Integrated Circuit] utilizado por Deep Blue para vencer a Kasparov. Cada circuito contenía un millón y medio de transistores y 480 circuitos como éste eran capaces de evaluar 200 millones de movimientos por segundo. Fuente: IEEE Chip Hall of Fame.

la marca Othello. Reversi, con 10^{28} posiciones diferentes y un árbol del juego con 10^{58} nodos, pondría a prueba la capacidad de un ordenador actual si quisieramos resolverlo por completo. Aun así, los campeones humanos se niegan a enfrentarse contra un ordenador porque el ordenador es demasiado bueno jugando a Reversi.

No en todos los juegos un ordenador es capaz de desmoralizar al jugador humano más competente. Hasta hace poco, eran los humanos los que se negaban a jugar contra un ordenador al Go... porque el ordenador era demasiado malo. El Go es un juego chino que consiste en rodear más territorio que el adversario sobre un tablero de 361 casillas (19x19), con 10^{170} posibles posiciones y un árbol de 10^{360} nodos con un factor de ramificación de 250. El número de posibilidades es tan enorme que las técnicas de búsqueda con adversario utilizadas en juegos tradicionales como las damas o el ajedrez, simplemente, no funcionan. Cambiando de estrategia, un equipo de Google DeepMind desarrolló AlphaGo¹⁴ recurriendo al uso de redes neuronales y técnicas de deep learning. Entrenado a partir de 160,000 partidas reales entre jugadores de Go y más de 30 millones de partidas que jugó contra sí mismo, AlphaGo fue capaz de derrotar en Londres a un jugador profesional de Go en octubre de 2015, el campeón europeo Fan Hui (dan 2), por 5-0. Para ello, AlphaGo utilizó nada menos que 1,202 CPUs y 176 GPUs. Sólo unos meses después, en marzo de 2016, una versión mejorada de AlphaGo venció en Seúl a Lee Sedol (dan 9), uno de los mejores jugadores de Go del mundo, por 4-1.

Y hasta ahora sólo hemos hablado de juegos perfectos: dos jugadores que intercalan sus movimientos en un juego en el que no interviene el azar y ambos disponen de información perfecta. Si añadimos el azar de los dados o la posibilidad de no disponer de toda la información relativa a la configuración del tablero (lo que los economistas denominan juegos con información imperfecta), la complejidad del problema aumenta:

- En el backgammon interviene el azar de los dados. En un tablero de sólo 28 posiciones, el juego admite 10^{20} posiciones (similar a las damas y menos que Reversi) pero el árbol del juego es de tamaño 10^{144} con un factor de ramificación de 250 (como el Go). Pese a la complejidad aparente del problema, el programa de ordenador BKG9.8 venció al campeón mundial de backgammon por 7-1 en julio de 1979, la primera vez que se logró algo así. A diferencia de los programas de ajedrez, que realizan búsquedas enormes, el programa BKG9.8 se basaba en el reconocimiento de patrones: la posición resultante de cada movimiento legal se evaluaba directamente, sin realizar ninguna búsqueda, por medio de funciones SNAC [*Smooth, Nonlinearity, and Application Coefficients*] que capturaban las características clave de cada configuración del tablero.¹⁵

¹⁴ David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, y Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. DOI: 10.1038/nature16961

¹⁵ Hans J. Berliner. Backgammon Computer Program Beats World Champion. *Artificial Intelligence*, 14(2):205–220, 1980. DOI: 10.1016/0004-3702(80)90041-7

- En el Stratego no se dispone de información perfecta. Con 10^{115} posiciones legales y un árbol de tamaño 10^{535} con un factor de ramificación medio de 21,739, su complejidad es muy superior a la del Go. Aunque existen campeonatos mundiales de Stratego, los programas de Stratego actuales no son mejores que un jugador humano de nivel intermedio.
- Juegos de cartas como el póker combinan el azar (al barajar las cartas) y la información incompleta (al desconocer las cartas de sus rivales). Aunque sin rival a la hora de poner cara de póker, el ordenador se encuentra con dificultades al explorar el espacio de búsqueda asociado a un problema con información asimétrica, ya que se ve obligado a evaluar un número enorme de posibilidades. Igual que sucedía con el Go, las estrategias tradicionales basadas en la poda alfa-beta no funcionan bien y sólo con la ayuda de redes neuronales se ha conseguido un programa de ordenador capaz de vencer a jugadores de póker profesionales en una variante llamada ‘heads-up no-limit Texas hold’em poker’ para dos jugadores: DeepStack.¹⁶ El nombre del programa hace referencia tanto al uso de técnicas de deep learning como a la jerga del póker, en la que ‘deep stack’ es el término utilizado para hacer referencia a una cantidad de dinero mayor que las apuestas sobre la mesa (o al jugador que dispone de tal cantidad de dinero).

En resumen, los ordenadores son capaces de vencer al ser humano en juegos con información perfecta como el ajedrez o el Go, pero sigue siendo extremadamente difícil enseñarles a jugar bien cuando otros factores (azar, información imperfecta) complican el juego.

Si los juegos de mesa han servido durante años como banco de pruebas para diferentes técnicas de Inteligencia Artificial (y para más de una proeza publicitaria), el desarrollo de videojuegos constituye otro sector en el que se demanda el uso de técnicas de Inteligencia Artificial, al menos sobre el papel. Aparte de su atractivo visual, que en el caso de los videojuegos AAA no tiene nada que envidiar a las superproducciones de Hollywood, el objetivo real de un videojuego es que no resulte excesivamente monótono para conseguir enganchar a sus usuarios, ni demasiado sencillo, ni demasiado difícil. Para ello, se recurre habitualmente a mecanismos¹⁷ relativamente sencillos que parecen dotar de inteligencia a los actores artificiales que intervienen en el juego. No son realmente inteligentes, aunque puedan parecerlo:

- En el clásico comeocos [Pac-Man], nuestro personaje tiene que ir recogiendo puntos amarillos a la vez que esquiva a sus enemigos, cuatro fantasmas llamados Blinky, Pinky, Inky y Clyde. Aunque en ocasiones pueda parecer que se coordinan de forma inteligente para rodearnos, en realidad cada uno de ellos emplea una sencilla heurística. Blinky, el fantasma rojo también conocido como Shadow [sombra], persigue a Pac-Man, nuestro personaje, buscando el camino más corto

¹⁶ Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, y Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017. ISSN 0036-8075. doi: 10.1126/science.aam6960

¹⁷ Ernest Adams y Joris Dormans. *Game Mechanics: Advanced Game Design*. New Riders Publishing, 2012. ISBN 0321820274

hasta nuestra posición actual. Pinky, el goblin rosa también llamado Speedy [rápido], intenta llegar lo antes posible a la posición que se encuentre cuatro pasos por delante de nosotros. Inky, el monstruo azul llamado Bashful [tímido], es algo más impredecible: se muestra inseguro, a veces nos evita pero, cerca de su hermano Blinky, se muestra más agresivo y nos persigue (en realidad, se limita a combinar nuestra posición y la de Blinky para decidir a dónde va). Por último, Clyde, el fantasma naranja, parece un tanto estúpido y va cambiando de dirección: nos persigue si está lejos pero se va a una esquina cuando estamos cerca, añadiendo algo de ruido. En definitiva, nada que no se pueda implementar buscando los caminos más cortos en el grafo asociado al mapa del juego. Aunque dé la sensación de que nos persiguen de forma colaborativa, no hacen más que utilizar estrategias complementarias, algo que difícilmente podríamos llamar Inteligencia Artificial,

- En los juegos de carreras, ya sean coches, motos, bicicletas, barcos o cualquier otro tipo de vehículo, es muy habitual que los personajes controlados por el ordenador “hagan la goma” como los ciclistas que se quedan descolgados del pelotón en cuanto empieza la subida a un puerto de montaña pero que no terminan de descolgarse del todo. Esta técnica, en inglés *rubber banding*, es una forma muy sencilla de adaptar la dificultad de un videojuego a la capacidad de un jugador. Una vez que superamos a nuestro oponente virtual, éste se limita a seguirnos a una distancia prudente, a la espera de un fallo nuestro que vuelva a darle algo de aliciente a la carrera. De paso, el motor del videojuego se ahorra algunos ciclos de reloj para determinar la estrategia de nuestro perseguidor, con lo que puede aprovechar la CPU para otros menesteres.

Estrategias similares son habituales para modelar el comportamiento, eminentemente reactivo, de multitud de personajes, tanto en videojuegos como en películas de animación. Y no sólo en películas de animación. Películas como *El señor de los anillos* [Peter Jackson, 2001] o *Avatar* [James Cameron, 2009] utilizan software como MASSIVE [*Multiple Agent Simulation System in Virtual Environment*] para simular batallas y el movimiento de multitudes de forma realista, en vez de recurrir a los miles de extras que se usaban en producciones como *Ben-Hur* [William Wyler, 1959] o *Braveheart* [Mel Gibson, 1995], que ya usaba imágenes creadas por ordenador [*CGI: Computer-Generated Imagery*] en algunas escenas.

Este tipo de técnicas, aunque en ocasiones se venden como Inteligencia Artificial, no se pueden considerar demasiado inteligentes. Al menos, salvo que nos limitemos a asumir que la inteligencia artificial, en minúsculas, tiene mucho de artificial y más bien poco de inteligente. ¿Qué características tienen los problemas cuya resolución sí que requiere genuinamente

de una inteligencia artificial?

Dada la definición que utilizamos de Inteligencia Artificial (la disciplina que se encarga de la resolución de tareas que, por ahora, los humanos realizamos mejor que las máquinas), muchos sistemas a los que nos hemos habituado, hace apenas unas décadas se considerarían ejemplos genuinos de I.A.: juegos de ajedrez, sistemas de reconocimiento de voz, asistentes digitales... Del mismo modo, muchas otras tareas para las que actualmente pensamos que requieren I.A. porque están reservadas a los humanos, en el futuro nos parecerá normal que las máquinas se encarguen de ellas. No por ello apreciaremos demasiada inteligencia en esas máquinas. Simplemente, serán capaces de realizar de forma eficiente las tareas para las que hayan sido diseñadas, ya sea jugar al ajedrez o conducir un coche.

Los ordenadores han demostrado, dada su potencia de cálculo, ser capaces de resolver fácilmente problemas que para nosotros resultan muy difíciles, desde problemas de cálculo numérico hasta el análisis de situaciones complejas que requieran evaluar millones de escenarios alternativos. Sin embargo, existen aún algunas tareas que a nosotros nos resultan sencillas pero que, sin embargo, para los ordenadores siguen siendo extremadamente difíciles. Sin duda, una posible causa de estas diferencias se debe a la evolución en sentidos opuestos que han experimentado los humanos, animales al fin y al cabo, frente a los ordenadores. Millones de años de evolución dotaron a los animales de capacidades sensoriales y motoras antes de que sólo los animales más evolucionados fuesen capaces de prever las consecuencias de sus acciones, razonar, comunicarse, utilizar herramientas o desarrollar el cálculo. En apenas unas décadas, los ordenadores, que comenzaron siendo herramientas de cálculo, se utilizaron luego para controlar herramientas, transmitir datos, razonar de forma simbólica y planificar. Sólo recientemente se ha conseguido dotar a las máquinas de capacidades sensoriales similares a las de animales y métodos, aún muy rudimentarios, de procesar el lenguaje natural mediante el que nos comunicamos los humanos.

Volvamos de nuevo nuestra mirada a la industria del entretenimiento, que no siempre anda tan descaminada como nuestro repaso inicial parecería indicar:

- En una escena con humor de *Terminator* [James Cameron, 1984], Arnold Schwarzenegger llega desde el futuro sin ropa y entra en un bar para encontrar algo de su talla y conseguir un medio de transporte adecuado. Su sistema visual procesa imágenes de la misma forma que lo haría un sistema de visión artificial. En primer lugar, se segmenta la imagen para detectar objetos, que pueden aparecer bajo diferentes condiciones de iluminación u ocluidos por otros objetos. Como resultado de este proceso de segmentación, se identifican las

fronteras de cada objeto y se procede a clasificarlos; esto es, identificar su tipo concreto.

- En películas como *Desafío total* [Total Recall, Paul Verhoeven, 1990] o *El quinto elemento* [The Fifth Element, Luc Besson, 1997] y en series de televisión como *El coche fantástico* [Knight Rider, 1982-1986], es habitual que nos encontremos con vehículos sin conductor. Aunque aún no se utilizan taxis voladores (o, al menos, no se llaman coches), los vehículos autónomos son una de las áreas de aplicación de la I.A. que más relevancia ha alcanzado durante la última década. En 2005, un vehículo llamado Stanley, diseñado por un equipo de la Universidad de Stanford, fue capaz de completar un gran desafío organizado por DARPA. Stanley recorrió 212 kilómetros (132 millas) por caminos de tierra del desierto de Mojave sin intervención humana, en algo menos de 7 horas (unos minutos menos que sus equipos rivales de la Universidad de Carnegie Mellon). Un par de años después, en 2007, DARPA organizó un nuevo gran desafío, esta vez en un circuito cerrado que simulaba un entorno urbano. En esta ocasión, venció el equipo de CMU, mientras que el de Stanford se tuvo que conformar con el segundo puesto. Desde entonces, empresas tecnológicas como Google (a través de su división Waymo), Uber o Tesla, así como casi todos los principales fabricantes de coches, han desarrollado prototipos de vehículos autónomos que ya circulan por carretera, incluso de noche o en condiciones de escasa visibilidad.

Estos coches autónomos incorporan múltiples sensores, como un LIDAR en el techo (*light detection and ranging*, una especie de radar que le permite al coche construir un mapa de 360º a su alrededor y medir distancias a los objetos que lo rodean), varios radares frontales y traseros (para medir distancias y velocidades de objetos más lejanos que los detectados por el LIDAR) y una cámara de vídeo en el parabrisas (para identificar el estado de los semáforos, situarse adecuadamente en su carril y detectar objetos móviles como peatones, ciclistas y otros vehículos), la cual puede ir complementada por una cámara de infrarrojos (para afinar la detección de objetos en condiciones de baja visibilidad). Para funcionar, además del hardware, el coche utiliza distintos tipos de técnicas en su software para resolver una variada gama de problemas prácticos:

- *Problemas de localización*

Grosso modo, la localización del coche se puede realizar con un sensor GPS. Sin embargo, el sistema GPS no ofrece la precisión necesaria para que el vehículo circule correctamente por su carril de la carretera. En combinación con el GPS, un coche autónomo suele utilizar un filtro de histograma. Este filtro, que usa las imágenes de

una cámara, le permite identificar los bordes del carril por el que circula y no salirse de él. En situaciones en las que no tiene acceso al GPS, p.ej. en un aparcamiento subterráneo, al vehículo no le queda más remedio que ir construyendo un mapa conforme explora el terreno usando SLAM [*Simultaneous Localization And Mapping*] e identificar su posición actual con la ayuda de un filtro de partículas. Por último, para saber cómo interactuar con objetos móviles, como peatones y ciclistas, debe identificar no sólo la posición de éstos, sino también estimar su velocidad, lo que consigue utilizando filtros Kalman con los que analiza la señal de vídeo de una cámara en el parabrisas del coche.

- *Problemas de navegación*

A la hora de planificar su ruta, un coche autónomo puede utilizar un algoritmo de búsqueda heurística como el algoritmo A*. Ahora bien, como los sensores del vehículo sólo le aportan información parcial con respecto a su entorno, la ruta planificada debe recalcularse varias veces por segundo en función de los datos captados por los sensores en cada momento. Piense, por ejemplo, en cómo actúa un conductor humano cuando busca un aparcamiento en un centro comercial y cómo rectifica su plan de actuación cuando no encuentra un hueco adecuado para su vehículo. A otro nivel de detalle, también pueden utilizarse otras técnicas, como la programación dinámica, para encontrar la ruta óptima de un punto a otro. Esto permite, por ejemplo, asociar un coste diferente a cada acción de tal forma que se minimizan los riesgos durante la conducción de un vehículo. Por ejemplo, si deseamos girar a la izquierda en una vía de doble sentido, algo que puede resultar peligroso en ocasiones, usando programación dinámica el coche puede ser capaz de decidir que resulta más aconsejable rodear la próxima manzana realizando tres giros a la derecha en vez de girar a la izquierda con tráfico de frente.

- *Problemas de control*

Curiosamente, ésta es la parte más simple de un coche autónomo. Para controlar su movimiento, un vehículo autónomo puede utilizar técnicas de control tradicional, como el control PID [*proportional-integral-derivative*], para regular su dirección y velocidad. Una estrategia de control que tiene sus orígenes en el siglo XIX, con el análisis del regulador centrífugo de las máquinas de vapor realizado por el físico James Clerk Maxwell en 1868 y que dio lugar a la teoría de control.

- Androides como C-3PO en *La guerra de las galaxias* y diversos dispositivos en otras películas de ciencia ficción sirven para traducir de un lenguaje a otro en tiempo real. Aunque pueda pensar que un intérprete

humano puede resultar imprescindible si viaja al extranjero, y es cierto que la calidad de las traducciones automáticas aún no alcanza la de un buen traductor humano, existen sistemas automáticos de traducción que pueden facilitar su comunicación en idiomas que no conoce. Desde traductores online a los que ya estará habituado y apps móviles que, tomando una instantánea de una señal o letrero, son capaces de traducir el mensaje de la señal de un idioma a otro, hasta auténticos intérpretes. Una aplicación móvil puede utilizar un proceso de reconocimiento óptico de caracteres [OCR: *Optical Character Recognition*] para extraer texto de una imagen, que posteriormente traduce de un idioma a otro y muestra sobreimpresionado sobre la imagen original. Más aún, desde hace unos años, el uso de redes neuronales permitió reducir la tasa de error de los sistemas de reconocimiento de voz. Una vez realizado el proceso de conversión de voz a texto, se puede traducir el texto y sintetizar una señal de voz que incorpore las mismas características vocales de la persona que esté utilizando el traductor [TTS: *Text-to-speech*]. Todo esto se puede hacer en tiempo real, tal como demostró en 2012 la división de investigación de Microsoft con un prototipo capaz de realizar las labores de un intérprete que traduce del inglés al chino.

- El lenguaje natural es muy rico en matices y ambigüedades, por lo que una máquina no siempre es capaz de interpretar adecuadamente un texto (ni, por tanto, traducirlo correctamente). Pese a ello, en 2011 se logró otro hito de la I.A.. Un ordenador de IBM llamado Watson fue capaz de vencer a los dos mejores concursantes de un concurso de televisión llamado *Jeopardy!*. La tecnología en que se basa Watson combina bases de datos enormes, técnicas de recuperación de información como las utilizadas en buscadores de Internet y técnicas heurísticas de evaluación de hipótesis candidatas que le permiten responder a las enrevesadas preguntas de ese concurso de televisión. Tras su mediática proeza, Watson dio lugar a una división de IBM que pretende explotar sus capacidades en diferentes dominios de aplicación. Por ejemplo, en un proyecto realizado en el hospital Memorial Sloan Kettering de Nueva York, Watson fue entrenado con 25 millones de documentos académicos publicados sobre el cáncer y, en una muestra de mil pacientes, fue capaz de realizar el mismo diagnóstico que los oncólogos en el 99 % de los casos y recomendar un tratamiento que se consideró mejor que el propuesto por los médicos en un 30 % de las ocasiones.

Michelle Zhou, que trabajó en el proyecto Watson de IBM Research antes de fundar una startup de análisis de sentimientos llamada Juji, divide la evolución de la I.A. en tres etapas. En la primera, inteligencia de reconocimiento [*recognition intelligence*], las máquinas son capaces

de identificar patrones y extraer información de bloques de texto, quizá incluso de derivar el significado de un documento a partir de su texto. En la segunda etapa, la inteligencia cognitiva [*cognitive intelligence*], las máquinas son capaces de realizar inferencias a partir de los datos: aprender de ellos. La tercera se alcanzaría cuando seamos capaces de crear seres humanos virtuales, que piensen, actúen y se comporten como lo hacemos nosotros. Algo así como el santo grial de la I.A.. Actualmente, aún estamos en la primera fase, en la que las técnicas de deep learning son cada vez más eficientes y mejores a la hora de descubrir patrones y sólo estamos empezando a vislumbrar las capacidades de la inteligencia cognitiva.

Como hemos visto, los primeros desarrollos de la I.A. se centraron en manipular símbolos de acuerdo a reglas lógicas, lo que les permitió realizar tareas como demostrar teoremas matemáticos, resolver puzzles u optimizar la distribución de componentes electrónicos en un circuito integrado (su layout). Otros problemas emblemáticos, como identificar objetos en imágenes o construir sistemas de reconocimiento de voz que transcriban una señal de audio en texto escrito, siempre se le resistieron a la I.A. simbólica. Sólo con la llegada de las técnicas de aprendizaje automático y el espectacular desarrollo del deep learning se ha conseguido construir sistemas que ofrezcan unos resultados comparables a los de un ser humano cuando resuelve este tipo de problemas.

Pero, antes de ver con más detalle en qué consisten el aprendizaje automático y el hoy llamado deep learning, o aprendizaje profundo, centremos nuestra atención en algunas cuestiones formales relacionadas con la resolución de problemas en I.A..

Técnicas heurísticas

Coinciendo con el invierno de la I.A., en los años 70 del siglo XX, se desarrolló una rama de la teoría de la computación conocida como complejidad computacional. Los estudios de complejidad computacional, básicamente, permiten clasificar los problemas de acuerdo a su dificultad inherente: su clase de complejidad. Además, establecen relaciones entre unas clases de complejidad y otras.¹⁸

En la práctica, una amplia gama de problemas se puede demostrar que son NP-completos, siendo los problemas NP-completos los problemas más difíciles que existen dentro de la clase de problemas computacionales conocida como NP. Formalmente, la clase NP engloba los problemas que pueden resolverse en tiempo polinómico utilizando una máquina de Turing no determinista. Ahora bien, los ordenadores digitales que utilizamos son máquinas de Turing deterministas y sólo son capaces de resolver en tiempo polinómicos los problemas de la clase P. La clase P se asocia a los problemas tratables: aquellos cuya solución se puede encontrar

¹⁸ Cristopher Moore y Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011. ISBN 0199233217

Para demostrar que un problema NP es NP-completo hay que realizar una reducción: mostrar que un problema NP-completo conocido se puede resolver si disponemos de una forma de resolver el problema cuya completitud queremos demostrar. Las demostraciones de completitud pueden no ser triviales. A menudo requieren razonar de forma analógica y transformar un tipo de problema en otro completamente diferente. Las reducciones más desafiantes demandan destreza, perspicacia y creatividad. O, como dirían algunos, una idea feliz.

en un tiempo razonable con la ayuda de un ordenador. La clase NP, en cambio, corresponde a los problemas cuya solución se puede verificar en un tiempo razonable: si nos dan la solución, podemos comprobar que es correcta. Sin embargo, para los problemas más difíciles de la clase NP, los problemas NP-completos, no se conoce ningún método eficiente que nos permita encontrar una solución.

La cuestión de si las clases P y NP coinciden o son diferentes aún no está resuelta y es clave en Informática teórica. De hecho, es uno de los problemas del milenio identificados por el Instituto Clay de Matemáticas. En la práctica, dado que casi todo el mundo asume que $P \neq NP$, esto implica que existen problemas que no pueden resolverse de forma eficiente con la ayuda de un ordenador... y nunca se podrá. Por tanto, a no ser que se demuestre que $P = NP$, lo cual es harto improbable, cuando se muestra que un problema es NP-completo, se está indicando que nadie puede esperar resolver ese problema de forma óptima en un tiempo razonable, incluso para problemas no demasiado grandes.

Ni siquiera aunque dispusiésemos de ordenadores cuánticos universales se podrían resolver los problemas NP-completos de forma eficiente.¹⁹ Hasta ahora, lo más que se ha logrado es una mejora polinómica en el tiempo de ejecución de algunos algoritmos. Por ejemplo, el algoritmo de búsqueda de Grover, propuesto por Lov Grover en 1996, permitiría obtener una mejora cuadrática al buscar una aguja en un pajar, por así decirlo. Se podría emplear para resolver cualquier problema NP en tiempo $O(2^{n/2} \text{poly}(n))$ frente al tiempo $O(2^n \text{poly}(n))$ que requeriría un algoritmo clásico que realizase una búsqueda exhaustiva en el espacio de soluciones del problema. En otras palabras, los ordenadores cuánticos podrían resolver problemas del doble de tamaño en el mismo tiempo que un ordenador convencional, pero no eliminarían el coste exponencial asociado a la resolución de problemas NP-completos.

Por tanto, dado que los problemas NP-completos requieren realizar una búsqueda en un espacio que crece de forma exponencial conforme aumenta el tamaño del problema que se pretende resolver y, no sólo nadie conoce cómo evitar esa búsqueda, sino que tampoco se espera que se consiga evitar, la única forma viable de resolver este tipo de problemas es recurrir a técnicas de tipo aproximado y heurístico, que son precisamente las técnicas que se suelen utilizar en Inteligencia Artificial. El mismo razonamiento se podría aplicar a una clase de problemas más general: todos los problemas NP-difíciles, que son al menos tan complejos como los más difíciles de la clase NP (los NP-completos).

De hecho, todos los problemas de los que se ocupa la Inteligencia Artificial son, desde el punto de vista formal, problemas NP-difíciles. Para estos problemas, si diseñamos un algoritmo que sea capaz de encontrar una solución aceptable de forma rápida y casi siempre correcta, podríamos considerar que el algoritmo es “inteligente”. La Inteligencia Artificial,

¹⁹ Scott Aaronson. *Quantum Computing since Democritus*. Cambridge University Press, 2013. ISBN 0521199565

Curiosamente, la mejora que ofrece el algoritmo de Grover es similar a la que puede ofrecer potencialmente la poda alfa-beta frente a la exploración completa del árbol minimax asociado a un juego. Si fuésemos capaces de ordenar la evaluación de los distintos movimientos de un juego para hacer la poda lo más efectiva posible, la complejidad de la búsqueda bajaría de $O(b^d)$ a $O(b^{d/2})$, donde b es el factor de ramificación del árbol y d es su profundidad. Con el mismo esfuerzo, la poda alfa-beta nos permitiría explorar un árbol el doble de profundo.

por tanto, se apropia del conocido aforismo de Voltaire: “lo perfecto es enemigo de lo bueno”. Las técnicas de I.A. buscan la excelencia sin obsesionarse demasiado con una perfección inalcanzable. Reconocen que pretenden abordar problemas computacionalmente intratables y que su búsqueda de soluciones está siempre sujeta a errores.

Una técnica heurística no garantiza que se encuentre la solución óptima de un problema, si bien puede ofrecer buenos resultados (o, al menos, mejores que los que se podrían obtener por otros medios). Una heurística no es más que un procedimiento que proporciona una solución aceptable a un problema mediante métodos que carecen de justificación formal. En cierto modo, su rol es similar al de un guía turístico cuando llegamos a una ciudad que no conocemos. El guía, si es bueno, nos llevará a los sitios más atractivos, aquéllos que potencialmente nos puedan interesar más. Aunque nadie nos garantiza que las recomendaciones del guía sean perfectas para nuestros gustos y aficiones, probablemente sean mejores que la alternativa de deambular sin rumbo en un entorno desconocido.

Las heurísticas se pueden diseñar de forma específica para resolver problemas concretos, como cuando se utiliza el algoritmo A* para obtener una solución de mínimo coste. Sin embargo, también existen heurísticas que se han mostrado tan útiles en una amplia gama de problemas que podríamos considerar heurísticas de propósito general. Estas heurísticas de propósito general se denominan metaheurísticas y se emplean mucho en la resolución de problemas complejos de optimización. De hecho, las metaheurísticas son, básicamente, algoritmos de tipo iterativo para resolver problemas de optimización. Usando una o varias soluciones candidatas, intentan mejorarlas con respecto a una medida de calidad dada utilizando estrategias de tipo heurístico.

Algunos ejemplos conocidos de metaheurísticas son el enfriamiento simulado [*simulated annealing*], la búsqueda tabú, los algoritmos GRASP [*Greedy Randomized Adaptive Search Procedure*], las nubes de partículas [*particle swarms*] o múltiples técnicas bioinspiradas, tales como los algoritmos genéticos, los algoritmos meméticos y los algoritmos basados en colonias de hormigas. Cuando los problemas de optimización combinatoria no son abordables con técnicas tradicionales como la ramificación y poda [*branch and bound*], las metaheurísticas pueden ser nuestro último recurso a la hora de encontrar soluciones satisfactorias para problemas como el de la mochila [*knapsack problem*], el del viajante de comercio [*TSP: Traveling Salesman Problem*], el de la ruta de los vehículos [*VRP: Vehicle Routing Problem*] o el de la asignación cuadrática [*QAP: Quadratic Assignment Problem*]. También pueden sernos de utilidad en problemas de optimización con restricciones como la programación entera, el caso discreto de la programación lineal.

Algunos expertos como Stuart Russell, de Berkeley, o Eric Horvitz, de Microsoft Research, consideran que la clave en el desarrollo de agentes in-

Además de, como asumimos con cierta naturalidad, dejarnos en la tienda de souvenirs que le concede a él la comisión más alta.

teligentes es precisamente la “optimalidad acotada” [*bounded optimality*]: la elección de un algoritmo que alcance un mejor compromiso entre su tiempo de ejecución y el error que comete al resolver un problema complejo. Abandonada toda esperanza de optimalidad, nos basta con encontrar soluciones suficientemente satisfactorias, en un tiempo suficientemente razonable, con la ayuda de técnicas heurísticas.

¿Acaso no recurrimos los seres humanos a criterios heurísticos a la hora de resolver los problemas de nuestro día a día? No sólo eso, sino que también incorporamos muchos sesgos cognitivos en nuestra toma de decisiones. Dichos sesgos, estudiados por la psicología y la economía del comportamiento, muestran las limitaciones de nuestro sistema cognitivo.²⁰ De forma sistemática, esos sesgos nos inducen a cometer errores de apreciación, siempre en el mismo sentido y, en ocasiones, en contra de nuestros propios intereses.

Tomemos como ejemplo a Harry Markowitz, que en 1990 recibió el Premio Nobel de Economía por el desarrollo de la teoría moderna de selección de cartera [*MPT: Modern Portfolio Theory*], una teoría de inversión que resuelve un problema de optimización multiobjetivo: maximizar el retorno de la inversión y minimizar su riesgo. La frontera de eficiencia de Markowitz es el conjunto de Pareto para su problema multiobjetivo: el conjunto de estrategias de inversión que alcanzan una máxima rentabilidad para un nivel de riesgo dado. Según la MPT, un inversor debería combinar valores en su cartera de forma óptima para maximizar su retorno dado un nivel de riesgo asumible por el inversor, un problema que se puede resolver matemáticamente una vez caracterizada la rentabilidad y volatilidad de los distintos tipos de activos en los que se puede invertir. Preguntado por cómo invertía sus ahorros de cara a su jubilación, el propio Markowitz reconocía que debería haber analizado la covarianza histórica entre clases de activos para obtener una frontera eficiente. Sin embargo, él mismo se imaginaba en el futuro lamentando, o bien no haber aprovechado las subidas de bolsa (si la estrategia óptima le recomendaba no invertir en acciones), o bien haber invertido demasiado en bolsa cuando ésta bajara. Dado que su objetivo en el fondo era minimizar sus remordimientos, decidió invertir un 50% en bolsa y un 50% en bonos. Se comporta de forma aparentemente irracional incluso la persona que pergeñó la teoría que establece la estrategia óptima que un inversor racional debería seguir. En la vida real, abandonó su propio modelo y se limitó a seguir una sencilla heurística. No porque no supiese resolver el problema de forma racional, sino porque consideró que una simple heurística era, para él, la solución más racional.

Puede que los atajos que tomamos a la hora de decidir cómo ahorraremos, a qué dedicamos nuestro tiempo o, en general, cómo vivimos, sean precisamente la estrategia más adecuada para tomar buenas decisiones en el complejo mundo real. Aunque se pueda pensar que dedicar menos

²⁰ Thomas Gilovich. *How We Know What Isn't So*. Free Press, 1993. ISBN 0029117062

En la selección de los activos de una cartera hay que considerar información sobre el comportamiento histórico de distintos tipos de activos. Esta información se utiliza para estimar las propiedades estadísticas de los distintos activos y un error de estimación puede cambiar significativamente la composición de una cartera y su riesgo asociado. Además, el pasado no siempre es útil para predecir el futuro, como suelen recordar los avisos del tipo ‘rentabilidades pasadas no garantizan rentabilidades futuras’. A no ser que tenga una confianza ciega en la información disponible, puede que resulte mejor ignorar por completo esa información. Una heurística del tipo 50%-50% no se ve afectada por los datos... ni tampoco pretende resolver ningún tipo de problema de optimización.

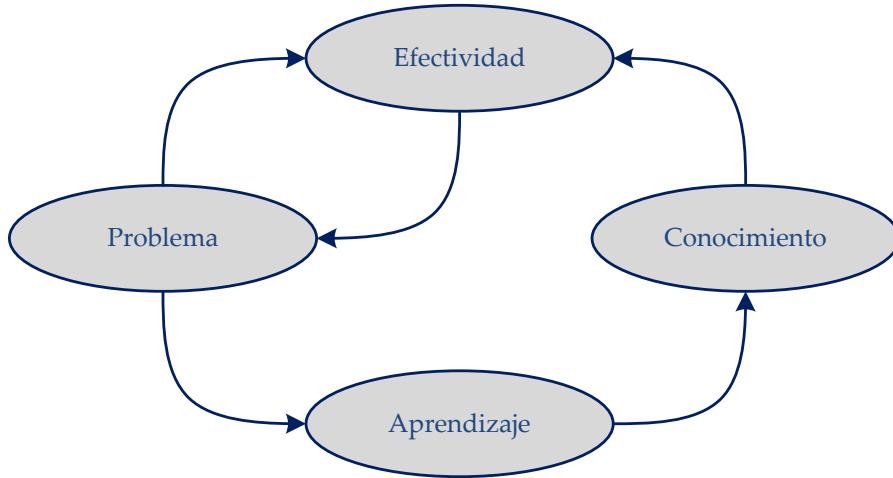
recursos computacionales a la resolución de un problema puede aumentar los errores que cometemos, el uso de heurísticas que favorezcan respuestas sencillas, tengan en cuenta menos factores y utilicen menos recursos computacionales puede llegar a ser ventajoso. En ocasiones, menos es más. Especialmente, si nuestros datos de entrada incluyen ruido y errores.

Recordemos un clásico de la Informática: GIGO [“*garbage in, garbage out*”], interpretado sarcásticamente como “garbage in, gospel out” cuando alguien confía ciegamente en los resultados obtenidos con un modelo computacional, que nunca deja de ser una simplificación de la realidad que pretende modelar.

Aprendizaje automático

Tradicionalmente, si alguien quería resolver un problema con la ayuda de un ordenador, tenía que diseñar e implementar un algoritmo que especificase, hasta el más mínimo detalle, qué es lo que el ordenador tenía que hacer. Para algunos problemas, con multitud de casos particulares imposibles de prever, esta estrategia, simple y llanamente, no funciona.

El aprendizaje automático, o *machine learning*, proporciona mecanismos mediante los cuales el ordenador es capaz de aprender por sí mismo a resolver un problema. En estos casos, el programador se encarga de diseñar un algoritmo de aprendizaje que resulte adecuado para el problema que se pretende resolver pero es el ordenador el que resuelve el problema, aprovechando para ello los datos a los que tenga acceso y las heurísticas de aprendizaje incorporadas en el algoritmo de aprendizaje creado por el programador. En cierto modo, el ordenador es capaz de programarse a sí mismo...



Si hay algo que distingue a la Inteligencia Artificial de otras ramas de la Informática es precisamente una de sus áreas clave: el aprendizaje automático o *machine learning*. Arthur Samuel, de IBM, definió el aprendizaje automático como el campo de estudio que dota a los ordenadores de la capacidad de aprender a resolver problemas para los que no han sido explícitamente programados. En Inteligencia Artificial, el aprendizaje se entiende como un proceso por el cual un ordenador es capaz de mejorar su habilidad en la resolución de un problema a través de la adquisición de conocimiento, conocimiento que obtiene a través de la experiencia.²¹

Figura 2: Interacciones entre el problema, el proceso de aprendizaje, el conocimiento adquirido y la efectividad observada en la resolución del problema. Adaptado de Pat Langley, “Elements of Machine Learning”, 1995, donde entorno y rendimiento ocupaban los lugares de problema y efectividad.

²¹ Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, 1st edition, 1997. ISBN 0070428077

Tal como lo definió Herbert Simon, en general, el aprendizaje denota cambios en un sistema que son adaptativos en el sentido de que permiten al sistema hacer la misma tarea, a partir de la misma posición, de un modo más efectivo. Cuando hablamos de aprendizaje automático en particular, se resaltan dos aspectos complementarios: el refinamiento de la habilidad en la resolución de problemas, lo que le permite al ordenador ser cada vez más efectivo, y la adquisición de conocimiento por medio de la experiencia. En ocasiones, el conocimiento adquirido será útil, no sólo para el sistema que aprende, sino también para nosotros. De hecho, fue el propio Simon el que, ya en 1969, propuso el uso del ordenador como fuente de conocimiento que ayudase a comprender mejor el comportamiento humano (mediante la realización de simulaciones, que era lo que se estilaba entonces).²²

¿Cómo puede adquirir experiencia un sistema automatizado? A partir de los datos que recibe. Un neurocientífico catalogaría como aprendizaje cualquier mecanismo que utilice las entradas presentes para dar forma a futuras respuestas; esto es, cualquier mecanismo que proyecte los datos de los que dispone en el presente hacia el futuro para cambiar su comportamiento. Ese cambio de comportamiento se realiza gracias a un proceso de entrenamiento con los datos disponibles, que le permiten a un sistema anticipar lo que puede suceder en el futuro. El entrenamiento puede realizarse de antemano (entrenamiento offline), de tal forma que el sistema se entrena antes de ponerlo en marcha en el entorno en el que desempeñará su labor. O también puede ir realizándose sobre la marcha (entrenamiento online), conforme el sistema realiza la tarea para la que ha sido diseñado. En el primer caso, su comportamiento se mantendrá estable a lo largo del tiempo, lo que nos permite predecir su rendimiento en un entorno controlado. En el segundo caso, su comportamiento se adaptará a los cambios que se produzcan en su entorno, lo que puede resultar impredecible.

Hasta hace no tanto tiempo se utilizaba el término “procesamiento de datos” para hacer referencia al uso de ordenadores en cualquier ámbito. De hecho, se sigue utilizando en los centros de datos que proporcionan la infraestructura necesaria para el *cloud computing*. Más recientemente, el procesamiento de datos se sustituyó por tecnologías de la información [*IT: Information Technology*], que básicamente se refiere a lo mismo pero implica un cambio de perspectiva. Se hace énfasis, no únicamente en el procesamiento de grandes cantidades de datos, sino en la extracción de información a partir de esos datos.

Los ordenadores, obviamente, sólo trabajan con datos: símbolos mediante los cuales se representan formalmente hechos, conceptos o instrucciones de forma adecuada para su comunicación, interpretación y procesamiento por seres humanos u otros medios automáticos. Sólo el ser humano les atribuye un significado a esos datos. Los datos son in-

²² Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, 3rd edition, 1996. ISBN 0262691914

Que se lo pregunten si no a los ingenieros de Microsoft, que crearon en 2016 un bot conversacional llamado Tay [*Thinking About You*]. Tay era capaz de aprender de sus interacciones con los usuarios humanos de la red social Twitter. Sólo diecisésis horas después de su puesta en marcha, el bot tuvo que ser retirado por estar completamente fuera de control...

formación en potencia, colecciones de símbolos en crudo que deben ser procesados para que sean significativos. La información la obtenemos nosotros asociando datos y dándoles significado en un contexto determinado.

La clave del éxito de los futuros sistemas de información residirá en que sean capaces de asociar datos provenientes de distintas fuentes sin que éstos muestren una conexión obvia, de forma que esa combinación nos proporcione beneficios. Tal como reconocía Roger Pressman a mediados de los años 90, la construcción de sistemas que extraigan conocimiento de los datos es uno de los desafíos más importantes de la Informática.²³ Ese no es, ni más ni menos, que el objetivo final del aprendizaje automático.

Para que el aprendizaje automático sea correcto, entendiendo éste como un proceso de generalización a partir de ejemplos concretos, hemos de disponer de suficientes casos de entrenamiento. Si las conclusiones obtenidas no vienen avaladas por un número suficiente de ejemplos, entonces la aparición de errores o ruido en los datos podría conducir al aprendizaje de un modelo erróneo, que no resultaría fiable en la práctica.

Cuantos más datos obtengamos, más fácilmente podremos diferenciar patrones válidos de patrones debidos a irregularidades o errores. Afortunadamente, hoy resulta relativamente fácil y barato recopilar grandes cantidades de datos relativos al problema que deseemos resolver, pues es sencillo digitalizar información y no resulta excesivamente caro almacenarla. Sin embargo, cuando el tamaño de los conjuntos de datos aumenta considerablemente, muchas de las técnicas más tradicionales de aprendizaje automático no resultan adecuadas por ser ineficientes y poco escalables. La necesidad de trabajar eficientemente con grandes conjuntos de datos dio lugar al desarrollo de la minería de datos o *data mining*^{24 25} a finales del siglo XX. La minería de datos, en cierto modo, no es más que el subconjunto de las técnicas de aprendizaje automático que se pueden aplicar a conjuntos de datos enormes. Para ello, las técnicas de aprendizaje se suelen combinar con métodos estadísticos y con técnicas de bases de datos que hagan posible el análisis de cantidades ingentes de datos, motivo por el cual, hoy en día, se suele hacer referencia a esta disciplina con el término genérico *big data*.

En la práctica, las técnicas de *data mining* se utilizan como parte de un proceso de extracción de conocimiento denominado KDD, acrónimo de *Knowledge Discovery in Databases*. Se entiende por KDD la extracción no trivial de información potencialmente útil a partir de un gran volumen de datos, en el cual la información está implícita pero no se conoce previamente.²⁶ El proceso de extracción de conocimiento incluye la preparación de los datos y la interpretación de los resultados obtenidos, para lo cual se requiere cierto conocimiento del dominio particular en el que se estén utilizando las técnicas de minería de datos. La denominada ciencia de datos [*data science*] es un término más reciente para hacer

²³ Roger S. Pressman. *Ingeniería del Software: Un enfoque práctico*. McGraw-Hill Interamericana, 1995. ISBN 8448100263

²⁴ Pang-Ning Tan, Michael Steinbach, y Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 1st edition, 2005. ISBN 0321321367

²⁵ Jiawei Han, Micheline Kamber, y Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 3rd edition, 2011. ISBN 0123814790

²⁶ Usama Fayyad, Gregory Piatetsky-Shapiro, y Padhraic Smyth. The kdd process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, November 1996. ISSN 0001-0782. DOI: 10.1145/240455.240464

referencia a este proceso en su conjunto y los llamados científicos de datos disfrutan de una de las profesiones más prometedoras en la actualidad (la mejor, según el portal de empleo Glassdoor).

Desde una perspectiva más académica, se denomina inteligencia computacional [*CI: Computational Intelligence*] a la capacidad de aprendizaje de un ordenador a partir de datos u observaciones experimentales. La inteligencia computacional engloba distintas metodologías y enfoques mediante los cuales se pretenden resolver problemas reales que no se pueden modelar matemáticamente, ya sea porque son excesivamente complejos para ser resueltos de forma analítica, porque presenten cierto grado de incertidumbre o porque son estocásticos y presentan cierto grado de aleatoriedad. Mediante este término paraguas, inteligencia computacional, se engloban diversas técnicas a las que habitualmente se les atribuye la etiqueta de *soft computing*: redes neuronales artificiales, computación evolutiva, modelos gráficos probabilísticos o lógica difusa [*fuzzy logic*]. Las técnicas de *soft computing* o inteligencia computacional están diseñadas para tratar de forma explícita con la presencia de imprecisión en los datos, incertidumbre, información incompleta y resultados aproximados. Como, en la práctica, esto es algo que sucede habitualmente, por no decir siempre, las técnicas de *soft computing* suelen ser especialmente robustas como técnicas de aprendizaje automático.

Aplicaciones del aprendizaje automático

El aprendizaje automático, por tanto, se basa en el uso de datos. Esos datos de entrenamiento se utilizan para obtener conclusiones más o menos generales partiendo de casos particulares. A diferencia del razonamiento deductivo, que va de lo general a lo particular, el razonamiento inductivo usado en el aprendizaje automático va de lo particular a lo general: un tránsito de las cosas individuales a los conceptos universales en palabras del propio Aristóteles. El razonamiento deductivo establece reglas que permiten establecer de forma precisa cuándo un razonamiento es válido: partiendo de las premisas, la conclusión se infiere necesariamente. Sin embargo, en el razonamiento inductivo no existe ningún tipo de garantía que nos indique cuándo un argumento es válido.

El filósofo inglés Francis Bacon, en su *Novum Organum* de 1620, identificó la inducción como la viga maestra sobre la que se asienta el método científico. En su *Tratado sobre la Naturaleza Humana* de 1739, el filósofo escocés David Hume formalizó la inducción como un proceso esencial para la adquisición de conocimiento en el mundo real, relegando la deducción al mundo de las ideas y las relaciones abstractas. Posteriormente, en sus *Principios de la Ciencia* de 1874, sería el lógico y economista inglés William Stanley Jevons el primero que interpretaría la inducción como la aplicación inversa de la deducción.

Alguno pensará, ingenuamente, que los datos nunca pueden reemplazar a la intuición humana. Más bien es al contrario: la intuición humana no puede nunca reemplazar a los datos. La intuición es a lo que uno recurre cuando carece de datos. Y dado que a menudo carecemos de los datos necesarios para tomar una decisión, la intuición es un bien muy valioso para nosotros. Pero, ¿y si tenemos frente a nosotros los datos necesarios? ¿Por qué no vamos a aprovecharlos?

Según una frase atribuida al pintor malagueño Pablo Ruiz Picasso, los ordenadores son inútiles, sólo proporcionan respuestas. Los ordenadores se limitan a ejecutar metódicamente los algoritmos con los que hayan sido programados, por lo que no son especialmente creativos. Sin embargo, si en vez de programarlos de la forma tradicional, indicándoles qué es lo que tienen que hacer en cada momento, los programamos con técnicas de aprendizaje automático, la situación cambia. Ahora serán capaces de realizar inferencias a partir de los datos que les lleguen y adaptar su comportamiento de acuerdo a ellos. En principio, cuantos más datos estén a su disposición, mayor puede ser su efectividad a la hora de resolver un problema dado.

El aprendizaje automático es lo que permite que un ordenador sea creativo. En vez de programar un ordenador para resolver un problema, se programa para que sea capaz de encontrar él mismo la forma de resolverlo. En ocasiones, hasta puede llegar a sorprendernos por sugerir soluciones que difícilmente se nos podrían haber ocurrido a nosotros. En cierto modo, los algoritmos de aprendizaje automático son algoritmos que construyen otros algoritmos, de forma que el ordenador se programa a sí mismo y nosotros no tenemos que hacerlo.

¿En qué situaciones es especialmente recomendable el uso de técnicas de aprendizaje automático? Básicamente, en tareas que nos gustaría poder automatizar pero que no hemos sido capaces de hacerlo (hasta ahora) por la complejidad que requeriría la programación de algoritmos tradicionales para resolver los problemas asociados a ellas. No se trata de problemas no resolubles, sino de problemas que nosotros resolvemos con relativa facilidad pero que son extremadamente difíciles de resolver utilizando un algoritmo secuencial diseñado manualmente.

Los problemas de reconocimiento de patrones son un ejemplo representativo del tipo de problemas para los que el aprendizaje automático resulta especialmente indicado. Los sistemas de reconocimiento de voz o de identificación de objetos en imágenes requieren tener en cuenta múltiples factores variables: tono [*pitch*], entonación, velocidad y ruido de fondo en una señal de voz; solapamientos, oclusiones, sombras, condiciones de iluminación y cambios de perspectiva en el caso de las imágenes. Esas variables dan lugar a un número tan enorme de situaciones posibles, que resultaría imposible de especificar de antemano. Sin embargo, si disponemos de datos, las técnicas de aprendizaje automático pueden

rescatarnos. En vez de diseñar un algoritmo a medida para resolver el problema, que siempre será frágil (además de muy costoso), diseñamos un algoritmo que aprenda de los datos disponibles y cree el “programa” necesario para resolver el problema. El “programa” generado no tiene por qué parecerse a un programa implementado manualmente. Por ejemplo, una red neuronal puede contener millones de parámetros ajustados numéricamente durante su proceso de entrenamiento y resultar completamente ininteligible para un ser humano. Nada que ver con un programa escrito manualmente, en un lenguaje de programación de alto nivel, como resultado del uso de prácticas recomendadas y de la aplicación de sólidos principios de diseño.²⁷

Si el algoritmo de aprendizaje tiene éxito, el programa creado automáticamente a partir de los ejemplos de su conjunto de entrenamiento, funcionará bien para nuevos ejemplos. El inconveniente, claro está, es que nunca podemos determinar del todo si ha sido correcta la generalización que se haya realizado a partir de los datos disponibles. Todo algoritmo de aprendizaje no es más que la descripción formal de una posible forma de generalizar a partir de ejemplos. Si los datos de los que disponemos en el conjunto de entrenamiento se parecen a los que nos encontraremos en el futuro, nuestras predicciones pueden ser acertadas. Si no, no nos queda más remedio que confiar en que los sesgos de tipo heurístico, que toda técnica de aprendizaje incorpora, serán los adecuados para resolver adecuadamente los problemas a los que nos tengamos que enfrentar. O, si detectamos una desviación en el rendimiento del programa que hemos aprendido, entrenarlo de nuevo para que se adapte mejor a la nueva situación.

La promesa del aprendizaje automático es, por tanto, hacer innecesario el desarrollo de algoritmos extremadamente sofisticados que requieran un esfuerzo manual immenseo para cada problema particular. ¿Cómo? Automatizando el proceso de resolución de los problemas mediante un proceso de aprendizaje, más o menos genérico, que permita resolver los problemas planteados suministrándole al ordenador conjuntos de datos. Eso sí, en ocasiones, los conjuntos de datos necesarios pueden llegar a ser enormes para obtener un rendimiento mínimamente aceptable.

Con ayuda de técnicas de aprendizaje automático, hoy utilizamos asistentes digitales que reconocen nuestra voz, no tenemos que pelearnos con el spam en nuestro buzón de correo porque previamente un filtro lo ha eliminado automáticamente, recibimos recomendaciones personalizadas de libros en Amazon o de películas en Netflix, existen sistemas de traducción automática de un idioma a otro, disponemos de aplicaciones para el móvil que nos dan el título y el intérprete de una canción que nos guste sólo con grabar un fragmento, tardamos sólo unos instantes en realizar búsquedas en el vasto océano de Internet y estamos a sólo un paso de desplazarnos de un lugar a otro utilizando coches completamente

²⁷ Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003. ISBN 0135974445

autónomos.

¿Dónde están los límites del aprendizaje automático? Los más pesimistas creen que la humanidad se encamina hacia la singularidad, el momento en el que máquinas superinteligentes comenzarán a mejorarse a sí mismas sin intervención humana. En el momento en el que superen la inteligencia humana, se generará un proceso de mejora exponencial tras el cual los humanos no seremos más que simples hormigas al lado de esas máquinas supuestamente superinteligentes. Más o menos, una situación al estilo de Skynet en Terminator... Sin embargo, por ahora lo que tenemos son termostatos inteligentes como Nest, con una inteligencia artificial muy reducida y centrada sólo en la resolución de problemas específicos. Eso es lo que nos ofrece el aprendizaje automático, nada parecido a una I.A. general que sustituya por completo al ser humano. Al menos, por el momento.

Sí podemos esperar resultados asombrosos a medio plazo. Basten algunos ejemplos recientes, que hasta hace nada eran pura ciencia ficción:

- En *2001: Una odisea en el espacio*, pudimos ver cómo HAL era capaz de aprender a leer los labios de los astronautas antes de empezar a tomar decisiones más drásticas. Hoy existen sistemas basados en redes neuronales que son capaces de leer los labios de un presentador de televisión si los entrenamos con un número suficiente de vídeos (miles de horas de programas de la BBC).²⁸
- En *Colossus: El proyecto prohibido*, el ordenador Colossus fue capaz de inventar un lenguaje nuevo para comunicarse con su homólogo soviético, *Guardian*. En cierto modo, se trataba de un lenguaje asentado sobre las mismas bases que nos permitirían comunicarnos con una civilización extraterrestre. La misma estrategia que se empleó para llenar el contenido de los discos de oro enviados en la sonda Voyager al espacio, lanzada en 1977 como una botella al océano cósmico por iniciativa del astrónomo Carl Sagan. De forma análoga, el traductor neuronal de Google²⁹ es capaz de traducir entre un par de idiomas para los que no ha sido previamente entrenado [*zero-shot translation*], sin utilizar otro idioma como paso intermedio. En cierto modo, es una señal de que ha sido capaz de desarrollar una especie de lenguaje mental [*mentalese*] o interlingua, como sostiene la hipótesis del lenguaje mental del filósofo americano Jerry Fodor.
- En *Juegos de guerra*, el ordenador W.O.P.R. (Joshua) fue capaz de establecer una analogía entre el juego de las 3 en raya [*tic-tac-toe*] y una guerra nuclear: ambos son ‘juegos’ en los que no se puede ganar, por lo que no tiene mucho sentido comenzarlos. El establecimiento de analogías es fundamental en el aprendizaje social de los animales. El aprendizaje social consiste en aprender comportamientos mediante la imitación de los comportamientos de otros individuos (p.ej. cuando

²⁸ Joon Son Chung, Andrew W. Senior, Oriol Vinyals, y Andrew Zisserman. Lip reading sentences in the wild. *arXiv e-prints*, arXiv:1611.05358, 2016. URL <http://arxiv.org/abs/1611.05358>

²⁹ Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, y Jeffrey Dean. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv e-prints*, 2016. URL <http://arxiv.org/abs/1609.08144>

un bebé aprende a mirar hacia donde apunta un dedo y no al dedo en sí, o cuando un perro es capaz de abrir una puerta sin necesidad de tener un modelo mental del mecanismo asociado a ella). A partir de demostraciones prácticas, algunos robots³⁰ ³¹ ya son capaces de aprender tareas que luego son capaces de reproducir por imitación, aunque sea de forma algo rudimentaria todavía en comparación con las capacidades de aprendizaje social de los animales más evolucionados.

Tipos de aprendizaje automático

Una primera clasificación de las técnicas de aprendizaje automático puede realizarse atendiendo a la filosofía utilizada en el proceso de adquisición del conocimiento:

- En el *aprendizaje supervisado* (o aprendizaje a partir de ejemplos, con profesor), los ejemplos de entrenamiento van acompañados de la salida correcta que el sistema debería ser capaz de reproducir. El entrenamiento de un modelo de aprendizaje supervisado consiste en ajustar sus parámetros para que sea capaz de reproducir una salida lo más parecida posible a la deseada. Una vez entrenado el modelo, lo verdaderamente importante es que sea capaz de generalizar correctamente. Esta capacidad de generalización consiste en que el modelo proporcione salidas adecuadas para datos de entrada diferentes a los datos utilizados durante su entrenamiento.

La familia de técnicas de aprendizaje supervisado engloba al aprendizaje memorístico [*rote learning*], a los modelos de aprendizaje por ajuste de parámetros y a una amplia gama de métodos de construcción de distintos tipos de modelos de clasificación, desde árboles de decisión y listas de decisión hasta máquinas de vectores de soporte SVM [*Support Vector Machines*] o redes neuronales utilizadas para clasificación. Tendremos ocasión de analizarlas con detalle más adelante.

- En el *aprendizaje no supervisado* (o aprendizaje por observación, sin profesor) se construyen descripciones, hipótesis o teorías a partir de un conjunto de hechos u observaciones, sin que exista información adicional acerca de cómo deberían clasificarse los ejemplos del conjunto de entrenamiento.

Será el método de aprendizaje no supervisado el que decida cómo han de agruparse los datos del conjunto de entrenamiento (en los métodos de agrupamiento o *clustering*) o qué tipo de patrones son más interesantes dentro del conjunto de entrenamiento (en las técnicas de extracción de reglas de asociación).

³⁰ Claudia Pérez-D'Arpino y Julie A. Shah. C-LEARN: Learning Geometric Constraints from Demonstrations for Multi-Step Manipulation in Shared Autonomy. En *Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA'2017)*, pages 4058–4065, 2017. ISBN 978-1-5090-4633-1. DOI: 10.1109/ICRA.2017.7989466

³¹ Tianhao Zhang, Zoe McCarthy, Owen Jow, Dennis Lee, Ken Goldberg, y Pieter Abbeel. Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. *arXiv e-prints*, arXiv:1710.04615, 2017c. URL <http://arxiv.org/abs/1710.04615>

Aprendizaje supervisado

El aprendizaje con profesor o aprendizaje supervisado, habitualmente asociado a los problemas de clasificación, es uno de los problemas más estudiados en Inteligencia Artificial. En un problema de clasificación, el objetivo de cualquier algoritmo de aprendizaje supervisado es construir un modelo de clasificación a partir de un conjunto de datos de entrada, denominado conjunto de entrenamiento, que contiene algunos ejemplos de cada una de las clases que pretendemos modelar.

Los casos del conjunto de entrenamiento incluyen, además de la clase a la que corresponde cada uno de ellos, una serie de atributos o características que se utilizarán para construir un modelo abstracto de clasificación. El objetivo del aprendizaje supervisado es la obtención de una descripción precisa para cada clase utilizando para ello los atributos incluidos en el conjunto de entrenamiento. El modelo que se obtiene durante el proceso de aprendizaje puede utilizarse para clasificar nuevos ejemplos (casos cuyas clases se desconozcan) o, simplemente, para comprender mejor los datos de los que disponemos, si nuestro modelo de clasificación es interpretable (algo que no siempre sucede).

Formalmente, un modelo de clasificación se puede definir de la siguiente manera. Si suponemos que todos los ejemplos que el modelo construido ha de reconocer son elementos potenciales de K clases distintas denotadas y_k , llamaremos $Y = \{y_k | 1 \leq k \leq K\}$ al conjunto de las clases de nuestro problema de clasificación. En determinadas ocasiones, extenderemos el conjunto de clases Y con una clase de rechazo y_0 que utilizaremos cuando el clasificador no sepa muy bien qué clase asignarle a un ejemplo dado. Esto es, asignaremos la clase de rechazo y_0 a todos aquellos casos concretos para los que no tengamos una certeza aceptable de que puedan ser clasificados correctamente usando alguna de las clases de Y . Al conjunto extendido de clases lo denotaremos $Y^* = Y \cup \{y_0\}$.

Un clasificador tradicional, o regla de clasificación, es una función $f : X \rightarrow Y^*$ definida sobre el conjunto de posibles ejemplos X tal que para todo ejemplo $x \in X$, el clasificador es capaz de asignarle una clase $f(x) = y \in Y^*$. Para construir ese clasificador, utilizaremos un conjunto de ejemplos de entrenamiento de la forma (x, y) , en los que x es un vector de características (los atributos que utilizaremos para determinar la clase de cada ejemplo) e $y \in Y$ es la clase a la que pertenece el ejemplo de entrenamiento. Una vez entrenado el clasificador, podremos utilizarlo para tratar de inferir la clase asociada a ejemplos x para los que no conoczamos su clase: $\hat{y} = f(x)$.

Los clasificadores no tienen por qué limitarse a indicar a qué clase pertenece un ejemplo. También pueden realizar una estimación de la probabilidad con la que un ejemplo puede pertenecer a cada una de las clases del problema. En este caso, el clasificador se puede ver como una

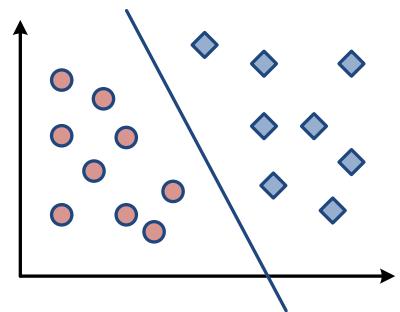


Figura 3: Aprendizaje supervisado: Un sencillo clasificador lineal es capaz de separar a la perfección dos clases diferentes (círculos rojos y rombos azules) siempre que las clases sean linealmente separables. Una línea recta, hiperplano en el caso general, define la frontera de decisión entre las dos clases.

Si las clases no fuesen discretas, en vez de clasificación, estaríamos hablando de regresión.

función $f : X \times Y \rightarrow [0, 1]$. Para un ejemplo dado, x , podemos estimar la probabilidad con la que el ejemplo pertenece a una clase y_k usando el clasificador: $\hat{p}(y_k|x) = f(x, y_k)$. Si lo que queremos es realizar una predicción concreta, simplemente nos fijamos en el máximo de la función f para cada una de las clases del problema dado un ejemplo x :

$$\hat{y} = \arg \max_{y_k} \hat{p}(y_k|x)$$

En determinados problemas, esta definición más flexible de un modelo de clasificación nos permitirá seleccionar el conjunto de hipótesis candidatas más probables. En este caso, el clasificador lo podemos utilizar como sistema de ordenación para establecer un *ranking*. En lo más alto del ranking se encontrarán las clases más prometedoras, por ser las más probables para un ejemplo dado. A continuación aparecerán, ordenadas de mayor a menor valoración, todas las demás clases de nuestro problema. Esta interpretación de los modelos de aprendizaje supervisado da lugar a múltiples aplicaciones relacionadas, que pueden verse como formas de aprendizaje supervisado. Es el caso de la recuperación de información, el procesamiento del lenguaje natural en los sistemas de pregunta-respuesta y, también, de los denominados sistemas de recomendación.

Pero antes de pasar a ver en qué consisten esas aplicaciones, mencionaremos un par de casos que merecen una atención especial:

- *Múltiples clases, jerarquías de conceptos y ontologías:*

En ocasiones, nuestro problema de clasificación puede tener un número enorme de clases diferentes y, además, esas clases, en lugar de ser disjuntas, puede que se solapen entre sí formando jerarquías de clases y ontologías. En una jerarquía de clases, algunas clases son especialización de otras, aunque sí suelen ser disjuntas las clases que se encuentran en el mismo nivel de la jerarquía de conceptos que define el conjunto de clases. En una ontología, no obstante, pueden coexistir múltiples jerarquías y existir solapamientos arbitrarios entre las distintas clases o conceptos recogidos en ella.

En tales situaciones, realizar una predicción concreta, de sólo una clase, puede ser extremadamente difícil y ni siquiera los seres humanos nos pondríamos de acuerdo en cuál debería ser la respuesta del clasificador. Por ejemplo, si queremos identificar qué objetos aparecen en una imagen, puede que nos baste con generar una lista de aquellos objetos que creemos que aparecen en la imagen, aunque no estemos del todo seguros de ello.

La salida del clasificador, por tanto, será una lista de las k clases más probables, el *top k*. Esa lista la podremos utilizar, a continuación, como entrada para otros módulos de nuestro sistema (siempre que

éstos sean capaces de trabajar con cierto grado de incertidumbre, claro está).

- *Clases poco balanceadas y detección de anomalías:*

En los problemas de clasificación, tradicionalmente, se supone que disponemos de ejemplos de distintas clases. Cada una de las clases está representada por un número similar de ejemplos y esto nos permite seleccionar modelos que resulten válidos para diferenciarlos. Sin embargo, en determinadas aplicaciones, unas clases son mucho más frecuentes que otras. Entonces es cuando hablamos de clasificación no balanceada.

Piense, por ejemplo, en un sistema de detección de intrusiones [*IDS: Intrusion Detection System*]. Lo habitual es que la mayor parte del tráfico de una red corresponda a un uso legítimo de la misma. Sin embargo, en ocasiones se producen ataques que nos gustaría poder detectar. Si los ataques corresponden únicamente al 0.01 % del tráfico habitual de la red, eso quiere decir que el 99.99 % restante es tráfico normal. Un clasificador por defecto, el que se limita a devolver siempre la clase más común, nos diría que todo el tráfico es normal... y acertaría el 99.99 % de las veces (algo que está muy lejos del rendimiento real de un sistema de aprendizaje en la mayoría de los casos). Sin embargo, pese a su tasa de acierto, sería completamente inútil para nuestro problema. Para que un clasificador no ignore las clases poco frecuentes, podemos sobreponerlas esas clases a la hora de construir nuestro modelo. Sin embargo, deberemos tener cuidado con que nuestro sistema no genere demasiados falsos positivos. Un sistema que genera falsas alarmas con frecuencia es, en la práctica, tan inútil como un sistema en el que nunca se dispara la alarma. En el primer caso, tenderemos a ignorar los avisos del sistema. En el segundo, viviremos felices en nuestra ignorancia.

Relacionado con el problema de la clasificación no balanceada está otro de los problemas habituales en minería de datos: la detección de anomalías.³² Una anomalía es una observación que se desvía tanto de las demás observaciones como para despertar sospechas de que se generó por un mecanismo diferente. Su detección es particularmente problemática porque a menudo es difícil distinguirlas del ruido presente en los datos. Y no siempre está claro cuándo una observación discordante es una anomalía y cuándo es, simplemente, ruido. Dependerá de la interpretación semántica que nosotros hagamos de los datos. De hecho, para detectar anomalías existen tanto técnicas supervisadas (de clasificación con clases muy poco balanceadas) como técnicas no supervisadas. En las segundas, las anomalías, denominadas *outliers* por los estadísticos, pueden verse, en ocasiones, como un subproducto de los métodos de *clustering*: las anomalías (o el ruido) son lo que

³² Charu C. Aggarwal. *Outlier Analysis*. Springer, 2013. ISBN 1461463955

queda al margen de los agrupamientos identificados por un método de *clustering*.

Consideremos ahora, con algo más de detalle, algunas aplicaciones del aprendizaje automático que, dada su importancia práctica, han dado lugar al desarrollo de técnicas específicas y a la formación de comunidades especializadas en la resolución de los problemas asociados a esas aplicaciones. Estamos hablando de la recuperación de información, del procesamiento del lenguaje natural y de los sistemas de recomendación.

Sistemas de recuperación de información

En determinadas ocasiones, la propia lista de hipótesis más probables nos puede servir para mostrársela directamente al usuario de nuestro sistema. Es lo que hacen los sistemas de recuperación de información [*information retrieval*].³³ Ante una búsqueda del usuario, generalmente especificada mediante una serie de palabras clave, el sistema le ofrece una lista ordenada de los documentos más relevantes para su consulta. Básicamente, estamos clasificando los documentos indexados por el sistema de recuperación de información en dos clases: los documentos que son relevantes para la consulta del usuario y los que no lo son. A diferencia de los clasificadores tradicionales, en los que las clases están predefinidas de antemano, en recuperación de información podríamos decir que las clases son las que cambian de una consulta a otra. Son los ejemplos los que siempre son los mismos: los documentos a los que el sistema de recuperación de información nos da acceso.

En la elaboración de la lista de resultados proporcionada como respuesta a una consulta, el sistema de recuperación de información puede tener en cuenta múltiples factores. Los buscadores web como Google o Bing emplean, no sólo el contenido de las páginas web que indexan, sino también información del usuario, como puede ser su localización. Si estamos buscando un buen restaurante de comida mexicana y nos encontramos en una ciudad como Granada, por ejemplo, no nos sería demasiado útil ver qué restaurantes de comida mexicana están abiertos en San Francisco.

Es habitual, también, que se tenga en cuenta la propia estructura de la web: como unas páginas incluyen enlaces a otras, podemos utilizar esa información para determinar cómo de importante es una página concreta con respecto a otras en las que aparecen términos similares. Eso es lo que hace el conocido algoritmo PageRank de Google, que jubiló a buscadores anteriores que se limitaban a indexar las páginas en función de su contenido y, por ese motivo, eran más vulnerables frente a manipulaciones de los resultados de búsqueda. ¿Alguien se acuerda de Altavista?

Buscadores como el de Google utilizan cientos de señales para es-

³³ ChengXiang Zhai y Sean Massung. *Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining*. ACM Books, Morgan & Claypool, 2016. ISBN 1970001194

stablecer el ranking de sus resultados, incluidas redes neuronales como RankBrain. Si Google encuentra una palabra o expresión con la que no esté familiarizada, utiliza RankBrain para adivinar qué palabras o expresiones pueden tener un significado similar, lo que le permite ser más efectivo frente a consultas a las que no se haya enfrentado antes. Según el propio Google, aunque se use en menos del 15 % de las búsquedas, RankBrain es el tercer factor más importante a la hora de ordenar los resultados de una búsqueda, sólo por detrás del contenido y los enlaces de la web.

Sistemas de pregunta-respuesta

Sistemas QA [Question Answering]³⁴ como IBM Watson, el utilizado en *Jeopardy!* para batir a concursantes humanos en televisión, están diseñados para responder preguntas formuladas en lenguaje natural. Aunque en el fondo se pueden ver como un tipo particular de sistemas de recuperación de información, normalmente requieren modelos algo más elaborados para representar una consulta que el simple conjunto de palabras clave empleado por los buscadores web. Los sistemas QA intentan construir una representación semántica de la pregunta. Esta representación nos puede ayudar a determinar el tipo de respuesta que debería proporcionar el sistema: ¿nos preguntan por una persona, por un lugar o por una fecha? Se trata de un problema de clasificación que también podemos resolver con la ayuda de técnicas de aprendizaje automático. Una vez determinado el tipo de respuesta requerido, podemos construir subsistemas especializados para cada tipo de pregunta, que extraigan información de las fuentes de datos relevantes y establezcan un ranking de las respuestas candidatas. En el caso de IBM Watson, el sistema original empleaba más de 50 componentes para evaluar cada hipótesis candidata, desde cómo encajaba la posible respuesta con la pregunta que se había efectuado hasta la fiabilidad de la fuente de datos de la que se había obtenido esa posible respuesta.

Dado que, actualmente, las máquinas no son capaces de interpretar correctamente todos los matices de la sintaxis y la semántica de un lenguaje humano, los sistemas QA se diseñan para resolver problemas en ámbitos muy delimitados. Algunos buscadores especializados incluyen características propias de este tipo de sistemas, como Wolfram Alpha (<https://www.wolframalpha.com/>) o el Knowledge Graph con el que Google complementa los resultados de una búsqueda utilizando información semántica recopilada de múltiples fuentes de datos. Mucho más habituales son los asistentes digitales, incorporados en multitud de dispositivos, desde teléfonos móviles y tablets hasta ordenadores personales y gadgets varios para el hogar. Las mayores empresas tecnológicas desarrollan asistentes virtuales como Siri de Apple, Cortana de Microsoft, M de

³⁴ Daniel Jurafsky y James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice-Hall, 2nd edition, 2009. ISBN 0131873210

Facebook, Alexa de Amazon o el Assistant de Google (tampoco es que se hayan heniado buscándole un nombre a su asistente). Diseñados para facilitar la interacción con el usuario utilizando el lenguaje natural, sólo han sido viables comercialmente gracias a las mejoras en los sistemas de reconocimiento de voz que se han logrado con el uso de redes neuronales artificiales.

Sistemas de recomendación

Los sistemas de recomendación [*recommender systems*]³⁵ son los que determinan qué mensajes leemos de nuestros contactos en una red social como Facebook, Twitter o LinkedIn. Nos sugieren qué artículos podríamos estar interesados en comprar en sitios de comercio electrónico como Amazon o en la página de resultados de un buscador como Google. Nos muestran las noticias que, tal vez, podrían interesarnos más. Personalizan nuestras experiencias musicales en iTunes o Spotify. Nos sugieren en Netflix qué película podríamos ver después de cenar. Incluso nos proponen en qué sitios podríamos cenar antes de ver la película recomendada. Dado su impacto en nuestra vida cotidiana, difícilmente encontraremos un tipo de sistema que pueda influir más en nuestra percepción del mundo que nos rodea.

Los sistemas de recomendación, igual que los buscadores, establecen un ranking con los ítems que más le podrían interesar a un usuario particular. Las técnicas utilizadas en los sistemas de recomendación actuales se podrían agrupar en dos grandes familias. En primer lugar, las técnicas de filtrado colaborativo utilizan las valoraciones de múltiples usuarios para proporcionar una recomendación. Por ejemplo, un sistema de este tipo podría recomendar artículos que otros usuarios con gustos similares a los nuestros ya han evaluado positivamente. En segundo lugar, los sistemas de recomendación basados en el contenido describen mediante atributos tanto a los usuarios del sistema como a los artículos que recomiendan. A continuación, utilizan esos atributos para realizar sus predicciones. Por ejemplo, si nos gustan las películas de suspense, un sistema así podría recomendarnos alguna novedad del mismo género, incluso aunque aún no haya sido evaluada por ningún otro usuario del sistema. Un sistema de recomendación, en la práctica, combinará múltiples técnicas para beneficiarse de la complementariedad que ofrecen y evitar las debilidades de las técnicas individuales, además de tener en cuenta el contexto para personalizar sus recomendaciones lo más posible (estacionalidad, localización geográfica...). Los sistemas sociales de recomendación también combinan información de nuestras redes sociales: nos sugieren quiénes podrían ser nuestros amigos (un problema conocido como predicción de enlaces en redes,³⁶) construyen modelos de importancia, influencia o confianza de nuestros contactos y aprovechan las etiquetas con las que,

Nada que ver, afortunadamente, con el infame Clippy de algunas versiones de Microsoft Office, que casi siempre decía “parece que está usted escribiendo una carta...”.

³⁵ Charu C. Aggarwal. *Recommender Systems: The Textbook*. Springer, 1st edition, 2016. ISBN 3319296574

³⁶ Víctor Martínez, Fernando Berzal, y Juan Carlos Cubero. A survey of link prediction in complex networks. *ACM Computing Surveys*, 49(4):69:1–69:33, 2017. DOI: 10.1145/3012704

colectivamente, clasificamos imágenes, vídeos y cualquier otro tipo de material online (dando lugar a las denominadas ‘folksonomías’).

Nada es gratis

Para estos y otros problemas de aprendizaje supervisado existen multitud de técnicas diferentes, que analizaremos más adelante. Algunas de esas técnicas destacan por su interpretabilidad, lo que puede resultarnos útil si tenemos que justificar el porqué de una decisión determinada. No es lo mismo decir que se tomó una decisión errónea porque utilizamos el ordenador como una caja negra, por lo que no sabemos la causa real del problema, que justificar racionalmente por qué se equivocó nuestro sistema de predicción, con datos concretos que avalen la decisión adoptada. Nuestra estabilidad laboral podría estar en juego. En otras ocasiones, nuestro principal objetivo es maximizar la precisión de nuestro clasificador, por lo que tal vez nos decantemos por técnicas que, aunque no destaque por su interpretabilidad, sean capaces de obtener mejores resultados cuantitativamente. Por último, cuando disponemos de conjuntos de datos gigantescos, nuestra principal preocupación posiblemente sea la utilización de técnicas escalables. La escalabilidad de una técnica particular hace referencia a que podamos aplicarla a conjuntos de datos más grandes invirtiendo proporcionalmente en recursos computacionales. Cualquier técnica que no sea escalable linealmente no será realmente aplicable a *big data* y requerirá del uso de técnicas de muestreo, con los consiguientes errores que ese muestreo podría introducir en la construcción de nuestro modelo de aprendizaje supervisado.

Sea cual sea la técnica por la que finalmente nos decantemos, sí deberíamos ser conscientes de algo: no existe ninguna técnica de aprendizaje supervisado que sea siempre mejor que otra. Es lo que se conoce con el nombre del teorema del “no almuerzo gratuito” [*no free lunch*], que en castellano podríamos traducir con un simple “nada es gratis”. David Wolpert, del Santa Fe Institute, lo enunció en 1996: promediado sobre todas las posibles distribuciones de datos, cualquier algoritmo de clasificación tiene la misma tasa de error cuando clasifica ejemplos no observados previamente.³⁷ Esto es, dadas dos técnicas de aprendizaje A y B, habrá tantas situaciones en las que A sea peor que B como situaciones en las que B sea peor que A.

Aunque ningún algoritmo es universalmente mejor que otro, afortunadamente, en el mundo real no nos encontramos con todas las posibles distribuciones de datos. Si realizamos suposiciones acerca de las distribuciones que nos podemos encontrar en el mundo real, podemos diseñar algoritmos de aprendizaje que funcionen bien sobre esas distribuciones (aunque no lo hagan en otras que, de todas formas, no son de nuestro interés). El objetivo del aprendizaje automático, por tanto, no es conseguir

A diferencia de la clasificación taxonómica formal que ofrece una ontología desarrollada por expertos como WordNet, las ‘folksonomías’ son sistemas de clasificación social o etiquetado colaborativo, con todas las imprecisiones y ambigüedades que ello conlleva.

³⁷ David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7): 1341–1390, 1996. ISSN 0899-7667. DOI: 10.1162/neco.1996.8.7.1341

un algoritmo de aprendizaje universal, que no existe, sino comprender qué aspectos son más relevantes en el problema concreto que pretendemos resolver y qué algoritmos o técnicas de aprendizaje automático pueden funcionar mejor bajo esas circunstancias.

Aprendizaje no supervisado

En las técnicas de aprendizaje supervisado, estimamos la probabilidad $\hat{p}(y_k|x)$ de que un ejemplo x pertenezca a cada una de las clases de nuestro problema, las cuales hemos de definir previamente. Nuestro clasificador es, en realidad, un modelo de la distribución de probabilidad condicionada $p(Y|X)$. Cuanto mejor sea dicho modelo, más útil nos resultará en la práctica.

En las técnicas de aprendizaje no supervisado, no existen clases predefinidas. Nuestro objetivo será encontrar patrones en los datos de entrada $x \in X$ que nos permitan construir un modelo de la distribución de probabilidad $p(X)$. ¿Qué utilidad puede tener algo así? El aprendizaje no supervisado nos puede servir como herramienta de análisis exploratorio de datos y para preprocessar los datos antes de utilizar una técnica supervisada.

- *Exploración de datos:*

Cuando nos llega un nuevo conjunto de datos pero aún no sabemos muy bien cuáles son sus características, las técnicas de aprendizaje no supervisado nos pueden servir para analizar su estructura. Con ayuda de técnicas no supervisadas, podemos descubrir qué patrones se repiten, cómo se agrupan los datos de forma natural y si existen algunas anomalías [*outliers*].

Con el apoyo de técnicas de visualización de datos, podemos inspeccionar visualmente los datos disponibles. En sistemas de información geográfica, los datos van asociados a coordenadas en un mapa, por lo que la visualización es inmediata. Las técnicas no supervisadas nos pueden ayudar en la construcción de mapas temáticos que representen la distribución geográfica de las variables que sean de nuestro interés. En marketing, las técnicas no supervisadas nos pueden ayudar a resolver problemas de segmentación de clientes. En recuperación de información, pueden servir para clasificar documentos sin tener que definir clases de antemano. En el análisis de web logs, la identificación de patrones de acceso similares nos puede servir para construir perfiles de los usuarios que acceden a un sitio web.

- *Preprocesamiento de datos:*

Más habitual resulta la utilización de técnicas no supervisadas como paso previo en la resolución de otros problemas de minería de datos,

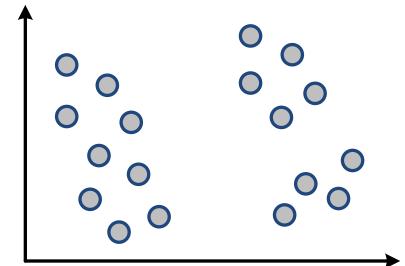


Figura 4: Aprendizaje no supervisado: Un algoritmo de agrupamiento será capaz de identificar las agrupaciones naturales presentes en un conjunto de datos, denominadas *clusters*. En este caso, ¿serán dos o tres los clusters identificados?

generalmente de tipo supervisado.

Por ejemplo, si tenemos un conjunto de datos con muchas variables pero las técnicas que nos gustaría utilizar no son demasiado escalables con respecto a la dimensionalidad de los datos, podemos utilizar técnicas no supervisadas para reducir su dimensionalidad.

En ocasiones, ciertas técnicas de aprendizaje supervisado no admiten determinados tipos de datos, como fechas o atributos numéricos, por lo que podríamos utilizar técnicas no supervisadas para transformar esos datos. Las técnicas de discretización hacen precisamente eso: convertir un atributo numérico continuo en un atributo categórico discreto.³⁸

Si un algoritmo de aprendizaje supervisado no es escalable con respecto al tamaño del conjunto de datos, las técnicas no supervisadas nos podrían servir para seleccionar un subconjunto de ejemplos representativos (muestreo o selección de instancias) o construir un conjunto de prototipos que representen las características originales de nuestro conjunto de datos completo. Entonces podríamos aplicar el algoritmo no escalable al conjunto de datos reducido, si bien ya no estaríamos haciendo *big data*, sino Estadística.

El rendimiento de una técnica de aprendizaje supervisada dependerá de la calidad del conjunto de entrenamiento con el que lo entrenemos. Aparte de preparar nosotros manualmente el conjunto de entrenamiento, podríamos recurrir a plataformas de crowdsourcing, como Amazon Mechanical Turk o los CAPTCHAs. En determinadas situaciones, no obstante, tal vez nos resulte demasiado costoso construir un conjunto de entrenamiento lo suficientemente grande. Aun así, si la recopilación de un conjunto de datos no etiquetado es factible, podemos realizar un proceso de pre-entrenamiento no supervisado [*unsupervised pretraining*]: utilizamos el conjunto de datos completo, no etiquetado, para intentar identificar las características más relevantes de nuestro conjunto de datos y, a continuación, afinamos nuestro modelo supervisado con la ayuda del conjunto de datos que sí tengamos etiquetado.

Clustering

La variante más conocida de las técnicas de aprendizaje no supervisado es la formada por los métodos de agrupamiento o *clustering*. Cuando un informático habla de *clustering* está haciendo referencia a lo que los especialistas en marketing denominan segmentación y los psicólogos llaman ordenación (que para los informáticos es algo muy diferente).

El objetivo de los algoritmos de *clustering* es encontrar agrupamientos de tal forma que los objetos de un grupo sean similares entre sí y diferentes de los objetos de otros grupos, a los que habitualmente llamaremos clusters.

Los resultados obtenidos por un método de agrupamiento dependen

³⁸ Huan Liu, Farhad Hussain, Chew Lim Tan, y Manoranjan Dash. Discretization: An enabling technique. *Data Mining and Knowledge Discovery*, 6(4): 393–423, 2002. ISSN 1573-756X. DOI: 10.1023/A:1016304305535

Un CAPTCHA [*Completely Automated Public Turing test to tell Computers and Humans Apart*] es una pregunta diseñada para determinar si un usuario es humano o no. A modo de test de Turing a la inversa, se emplea para evitar spam en muchas aplicaciones web. El test le pide al usuario que demuestre que es humano respondiendo a una pregunta que resulta sencilla para nosotros pero que sería muy difícil para un ordenador y puede ir acompañado de un reCAPTCHA, una segunda pregunta en la que se le pide que resuelva un caso aún no resuelto, con lo que realmente está proporcionando una etiqueta que se podrá utilizar como ejemplo del conjunto de entrenamiento.

de varios factores: el algoritmo de agrupamiento empleado, el conjunto de datos disponible y la medida de similitud utilizada para comparar objetos. La medida de similitud, usualmente, se define como una medida de distancia. Una distancia es, realmente, una medida de disimilitud: $d(x_i, x_j) > d(x_i, x_k)$ nos indica que el objeto x_i es más parecido a x_k que a x_j . La definición de la métrica de similitud (o distancia) hay que diseñarla de forma específica para cada problema de agrupamiento con el que nos encontremos. Será distinta en función de la interpretación semántica que nosotros hagamos de los datos.

¿Cuál es la forma natural de agrupar los datos? No existen respuestas absolutas, el *clustering* siempre será subjetivo. En general, ¿qué es lo que se entiende por un buen agrupamiento? Normalmente, aquel agrupamiento en el que cada cluster contiene ejemplos muy similares y en el que los ejemplos de distintos grupos no son similares entre sí. En términos de distancias, pretendemos minimizar la distancia intra-cluster (dentro de un mismo cluster) y maximizar la distancia inter-cluster (entre clusters diferentes).

El algoritmo que utilicemos para agrupar los datos puede tener un gran impacto en los resultados finales que obtengamos. El algoritmo perfecto, que no deja de ser una utopía, debería ser capaz de manejar atributos de distintos tipos (tanto numéricos como categóricos), requerir un número mínimo de parámetros, ser altamente escalable, tolerar la presencia de ruido y *outliers*, identificar clusters de formas arbitrarias, proporcionar resultados que fuesen independientes con respecto al orden de presentación de los patrones utilizados en su entrenamiento, funcionar correctamente en espacios con muchas dimensiones diferentes, incorporar restricciones especificadas por el usuario [*domain knowledge*] y generar resultados interpretables. Obviamente, ningún algoritmo real cumple con los requisitos de esta lista de los Reyes Magos.

Los métodos de agrupamiento más utilizados en minería de datos se podrían clasificar en las siguientes familias: métodos particionales, métodos jerárquicos, métodos basados en densidad y *clustering* en subespacios. Veamos, a continuación, en qué consiste cada una de ellas.

- *Métodos de agrupamiento por particiones*

En los métodos de agrupamiento de este tipo, generalmente se utiliza un parámetro k que indica el número de clusters que deseamos encontrar en los datos. Puede que se trate de un número de clusters ya conocido, fijado de antemano, o que tengamos que probar con distintos valores de k hasta encontrar el número más adecuado de clusters para nuestro conjunto de datos.

El algoritmo más conocido, el algoritmo de las k medias o *k-means*, asocia un centroide a cada cluster. Este centroide, que corresponde al centro geométrico del cluster, hace que sólo podamos encontrar

clusters globulares con este algoritmo. Inicialmente, se pueden escoger k ejemplos aleatoriamente para inicializar los centroides (aunque también hay métodos más sofisticados de inicialización). Los puntos correspondientes a los distintos ejemplos se asignan al cluster cuyo centroide esté más cerca en cada momento (utilizando cualquier métrica de distancia que resulte adecuada para nuestro problema). Iterativamente, el algoritmo recalcula los centroides en función de las asignaciones de puntos a clusters y reasigna los puntos a los clusters más cercanos, hasta que llega un momento en el que los centroides dejan de cambiar. Si consideramos el número de puntos n del conjunto de entrenamiento, el número de clusters k , el número de atributos d y el número de iteraciones i , obtenemos un algoritmo de eficiencia $O(n * d * k * i)$, lo suficientemente buena para poder aplicarse en grandes conjuntos de datos.

Otros algoritmos recurren al uso de medianas en lugar de medias, con el objetivo de limitar la influencia de los *outliers*. Es el caso del algoritmo de los k medoides [*k-medoids*], también conocido como PAM [*Partitioning Around Medoids*]. Por desgracia, el cálculo de medianas aumenta el coste computacional del algoritmo de agrupamiento. Por este motivo, se han propuesto múltiples variantes para mejorar su eficiencia, tales como CLARA [*Clustering LARge Applications*] o CLARANS, una versión de CLARA que utiliza búsqueda aleatoria.

Algoritmos como el de las k medias sólo funcionan con atributos de tipo numérico. Si queremos trabajar con datos de tipo categórico, tendremos que cambiar la forma de calcular los centroides de los clusters. Por ejemplo, el algoritmo de las k modas [*k-modes*] utiliza modas en lugar de medias, lo que nos permite seleccionar prototipos para cada clúster cuando tenemos atributos de tipo categórico, en los que la media aritmética no tiene sentido.

Los lectores con una formación más matemática puede que hayan oído hablar del *clustering* espectral [*spectral clustering*].³⁹ Su idea básica consiste en representar el conjunto de datos en forma de grafo derivado de una matriz de similitud. Utilizando alguna métrica adecuada que evalúe el parecido entre dos ejemplos dados, la matriz de similitud recoge el parecido de todos los ejemplos con todos los demás. Si disponemos de n ejemplos, la matriz de similitud será de tamaño $n \times n$. Las técnicas de *clustering* espectral cuantifican la calidad de un clúster como una función de ‘corte’ que separa al clúster del resto de la red, motivo por el que es habitual su uso en detección de comunidades (el problema del agrupamiento aplicado a redes). ¿Cómo lo consiguen? En primer lugar, se transforma el conjunto de nodos del grafo en un conjunto de puntos en un espacio métrico cuyas coordenadas corresponden a los k vectores propios más relevantes de la matriz

³⁹ Ulrike von Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17(4):395–416, 2007. DOI: 10.1007/s11222-007-9033-z

laplaciana del grafo, de ahí su denominación. Una vez hecho esto, se pueden agrupar los puntos resultantes de esa transformación utilizando cualquier método de agrupamiento por particiones, como el algoritmo de las k medias.

- *Métodos de agrupamiento jerárquico*

Los algoritmos de agrupamiento jerárquico construyen un árbol en el que se representa el parecido entre distintos objetos, igual que en una taxonomía. Este árbol se denomina dendrograma. Las hojas del árbol corresponden a los ejemplos de nuestro conjunto de datos y la altura a la que se unen en el árbol representa la similitud entre dos objetos: los dos agrupamientos que se combinan en ese nodo del árbol. La visualización de un dendrograma nos puede ayudar a determinar el número adecuado de agrupamientos en un problema de agrupamiento, aunque no siempre será demasiado fácil. También nos puede servir en herramientas interactivas de exploración de datos.

En función de cómo se construye el dendrograma, existen dos tipos de técnicas de *clustering* jerárquico. Las técnicas aglomerativas construyen el dendrograma desde sus hojas hacia la raíz. Comienzan considerando cada ejemplo como un cluster individual. En cada iteración del algoritmo, se combina el par de clusters más cercano. Y así, hasta que sólo quede uno, como en *Los inmortales* (o k , si sabemos el número de clusters que estamos buscando en los datos). Las técnicas divisivas, en cambio, construyen el dendrograma comenzando en su raíz. Se empieza con un único cluster que engloba todos los casos de nuestro conjunto de datos. En cada iteración, se divide en dos uno de los clusters hasta llegar a las hojas del dendrograma, en las que cada cluster contendrá un único caso. Obviamente, podemos detener el proceso en cuanto alcancemos el número de clusters requerido.

El principal inconveniente del *clustering* jerárquico es su baja escalabilidad. Tal cual, los métodos jerárquicos son, al menos, cuadráticos: $\geq O(n^2)$. Esto los hace inviables para conjuntos de datos grandes. En ocasiones se emplean para estimar un valor adecuado para el parámetro k que representa el número de clusters deseado en algoritmos como el de las k medias, aplicando el método jerárquico sobre una muestra de los datos y no sobre el conjunto de datos completo.

A diferencia de algoritmos como el de las k medias, que sólo son capaces de identificar clusters de forma globular, los métodos jerárquicos son muy flexibles a la hora de identificar clusters de formas arbitrarias (podemos definir la función de similitud que queramos a la hora de construir el dendrograma). Esto ha dado lugar al desarrollo de algoritmos jerárquicos más eficientes, aunque no siempre demasiado escalables, con nombres como BIRCH (del que el método de *clustering* en dos pasos de SPSS es una variante), ROCK, CURE o Chameleon.

Muchos métodos de detección de comunidades en redes también

En Matemáticas, el espectro de una matriz es su conjunto de valores propios [*eigenvalues*], asociados cada uno de ellos a un vector propio [*eigenvectors*].

son de tipo jerárquico. Es el caso del método de Newman y Girvan, que está basado en una medida de intermediación [*edge betweenness*], o del método de Radicchi, que utiliza un coeficiente de agrupamiento y resulta más eficiente que el método de Newman y Girvan. Ambos son algoritmos jerárquicos divisivos. También existen métodos aglomerativos de detección de comunidades, usualmente basados en alguna medida numérica de modularidad Q , mediante la que se convierte el problema de detección de comunidades en un problema numérico de optimización.

■ *Métodos de agrupamiento basados en densidad*

Las técnicas de agrupamiento basadas en densidad son una alternativa eficiente a los métodos jerárquicos si deseamos encontrar clusters de formas irregulares; esto es, agrupamientos no globulares. Para ellas, un cluster es una región densa de puntos, separada por regiones poco densas de otras regiones densas. Los algoritmos basados en densidad son capaces de detectar de forma muy eficiente clusters de este tipo (repetimos el trabalenguas: regiones densas de puntos separadas de otras regiones densas por regiones poco densas). Además de ser escalables, funcionan bien con clusters cuando estos están entrelazados y son robustos ante la presencia de ruido y *outliers* en los datos.

¿Por qué son escalables? Básicamente, porque utilizan un criterio de agrupamiento local para ir conformando los agrupamientos. En un único recorrido del conjunto de datos son capaces de agruparlos en clusters, con la única ayuda de una estructura de datos que sirva de índice para acceder de forma eficiente a los ejemplos en el entorno de un caso dado.

Algunos ejemplos representativos de este tipo de algoritmos son DBSCAN [*Density Based Spatial Clustering of Applications with Noise*], OPTICS [*Ordering Points To Identify the Clustering Structure*] o DENCLUE [*DENSity CLustErIng*]. También existen métodos basados en densidad que, en vez de medir distancias, calculan la similitud entre dos ejemplos en función del número de vecinos que comparten [SNN: *Shared Nearest Neighbor*], si bien son menos escalables que los anteriores.

Existen otras muchas técnicas de agrupamiento. Algunas, de carácter probabilístico, intentan ajustar los datos a un modelo matemático previamente escogido, haciendo la suposición de que los datos observados provienen de la superposición de varias distribuciones de probabilidad. Es el caso del algoritmo EM [*Expectation Maximization*] que se utiliza para ajustar los parámetros de un modelo GMM [*Gaussian Mixture Model*].

Hay algo que debemos tener muy presente cuando utilizamos cualquier técnica de agrupamiento. Tal como dice el aforismo atribuido a Ronald Harry Coase, un economista británico de la Escuela de Chicago que

El *betweenness* se utiliza como medida de importancia de un nodo o un enlace en una red midiendo el número de caminos mínimos que pasan por cada nodo o enlace. El coeficiente de agrupamiento es una medida local del número de ciclos (triángulos para ser exactos) en los que está involucrado un nodo. Los enlaces que conectan dos comunidades diferentes tendrán un *betweenness* elevado, mientras que estarán involucrados en menos ciclos que los enlaces que conectan nodos de la misma comunidad (asumiendo que una comunidad contiene nodos muy interconectados entre sí).

recibió el premio Nobel de Economía en 1991: “si se tortura lo suficiente a los datos, éstos acaban confesando”. Los algoritmos de detección de agrupamientos están diseñados para encontrar clusters. Lo hacen, incluso, cuando nuestro conjunto de datos es completamente aleatorio (puede comprobarlo usted mismo). Da igual que utilicemos métodos particionales como el de las k -medias, algoritmos jerárquicos o métodos basados en densidad como DBSCAN. Si nos empeñamos en encontrar clusters, los encontraremos. Aunque no estén ahí.

¿Existe alguna limitación inherente a cualquier método de agrupamiento que utilicemos? Sí, existe un teorema de imposibilidad asociado a los problemas de agrupamiento. No es tan llamativo como el teorema de imposibilidad de Arrow, pero tiene sus implicaciones para nosotros. Lo formuló Jon Kleinberg en 2002 y establece que, aunque existen múltiples algoritmos que cumplen dos propiedades, ninguno puede cumplir simultáneamente las tres propiedades siguientes:⁴⁰

- *Invarianza frente a cambios de escala:* Si transformamos los datos cambiando su escala, de forma homogénea en todas direcciones, el resultado del algoritmo de agrupamiento no debería cambiar.
- *Consistencia:* Si estiramos los datos de forma que las distancias entre los clusters aumenten, o los comprimimos haciendo que las distancias dentro de un cluster disminuyan, los resultados proporcionados por el algoritmo de agrupamiento no deberían cambiar.
- *Riqueza:* Un algoritmo de agrupamiento debería ser capaz, teóricamente, de producir cualquier partición arbitraria del conjunto de datos (antes de conocer la distancia entre parejas de ellos, que es la que se emplea para ajustar la partición al conjunto que nos den).

El teorema de imposibilidad de Kleinberg es, para las técnicas no supervisadas, análogo al teorema de Wolpert para las técnicas de aprendizaje supervisado. En su formulación, es también similar a otro teorema muy conocido: el teorema CAP de Eric Brewer. El teorema CAP se aplica al almacenamiento de datos en sistemas distribuidos, p.ej. en bases de datos NoSQL. En estos sistemas es imposible ofrecer simultáneamente las tres garantías siguientes: consistencia (C), disponibilidad (A) y tolerancia a particiones (P). Se pueden escoger dos de las tres, pero no las tres, lo que da lugar a sistemas CA (como los tradicionales gestores de bases de datos relacionales, sistemas centralizados que no son tolerantes a particiones), a sistemas AP (Cassandra y CouchDB son sistemas NoSQL que sacrifican eventualmente la consistencia para garantizar su disponibilidad) y a sistemas CP (MongoDB y Hbase son sistemas NoSQL que optan por sacrificar provisionalmente la disponibilidad de los datos para garantizar su consistencia).

⁴⁰ Jon Kleinberg. An Impossibility Theorem for Clustering. En *Proceedings of the 15th International Conference on Neural Information Processing Systems, NIPS'02*, pages 463–470, 2002

Kenneth Arrow, premio Nobel de Economía en 1972, se hizo famoso por mostrar que, cuando hay más de dos candidatos alternativos, no existe ningún sistema de votación que cumpla simultáneamente unos mínimos que podrían parecer razonables: (1) cuando todos prefieren a X frente a Y, saldrá elegido X antes que Y; (2) si se conservan las preferencias entre X e Y, modificaciones en las preferencias entre otros pares (como X y Z, Y y Z, o Z y W) no modificarán el resultado; y, (3) no existe un dictador, alguien que con su voto sea capaz de determinar el resultado de la votación.

La consistencia hace referencia a que cualquier operación de lectura proporciona el valor más reciente de un dato. Disponibilidad [*availability*] indica que cualquier solicitud recibe siempre una respuesta. Tolerancia a particiones es que el sistema pueda seguir funcionando aunque se pierda la conexión entre algunos nodos del sistema distribuido.

A parte de las limitaciones generales de cualquier método de agrupamiento, cuando tenemos datos con muchas variables, la dimensionalidad de los datos hace que las medidas de similitud o distancia empleadas por los algoritmos de *clustering* tradicionales dejen de ser útiles. Conforme va aumentando el número de dimensiones, llega un momento en el que todos los ejemplos de nuestro conjunto de datos están igual de lejos unos de otros. Este es el motivo por el que se desarrollaron métodos de agrupamiento en subespacios [*subspace clustering*], que buscan clusters utilizando distintas combinaciones de atributos (subespacios del espacio original en el que se encuentran los datos). Es el caso de algoritmos como CLIQUE [*CLustering In QUEst*], del proyecto Quest de IBM que dio lugar al desarrollo de las técnicas de minería de datos en los años 90, o PROCLUS [*PProjected CLUSTering*]. Estos algoritmos, internamente, aprovechan las técnicas que se desarrollaron para encontrar patrones frecuentes en bases de datos, de las que hablaremos brevemente a continuación.

Reglas de asociación

Si hay un problema que resalte las diferencias entre las técnicas de minería de datos y los métodos tradicionales de aprendizaje automático, ése es el de extracción de reglas de asociación. En los años 90, multitud de empresas comenzaron a recopilar grandes cantidades de datos pero no disponían de las herramientas necesarias para poder extraer patrones útiles de esas grandes bases de datos. IBM lanzó un proyecto de análisis de datos en su Almaden Research Center, sito en el valle del mismo nombre en San José, California. En ese proyecto, denominado Quest, se desarrollaron las técnicas de identificación de patrones y extracción de reglas de asociación que darían lugar a la minería de datos.

Una regla de asociación es un modelo muy simple de representación del conocimiento que puede ser útil para caracterizar las regularidades que se pueden encontrar en grandes bases de datos. Una regla de asociación es de la forma $X \Rightarrow Y$, donde tanto X como Y son conjuntos de ítems, llamados *itemsets*, de intersección vacía; esto es, sin ítems en común. Tradicionalmente, las reglas de asociación se han usado para analizar bases de datos transaccionales, en las que una transacción es un conjunto de artículos o ítems (p.ej. los productos incluidos en la cesta de la compra de un cliente). La interpretación intuitiva de una regla de asociación $X \Rightarrow Y$ es que las transacciones que contienen a X también tienden a contener a Y . Para evaluar las reglas de asociación se suelen emplear dos medidas: su confianza (la proporción de las transacciones que, conteniendo a X , también incluyen a Y) y su soporte (la fracción de transacciones en la base de datos que contienen tanto a X como a Y).

Existe una leyenda urbana, de origen incierto, que cuenta que, ana-

lizando las ventas de un supermercado, se descubrió que los jueves por la tarde aumentaban las ventas conjuntas de pañales y latas de cerveza. Este tipo de información le puede servir al gerente del supermercado para decidir qué ofertas realizar: podría ofrecer cervezas con descuento a quien comprase pañales. Bueno, mejor no. Algo así quedaría fuera de lo que actualmente se considera políticamente correcto y, posiblemente, sólo conseguiría una denuncia (¡un supermercado promueve la irresponsabilidad de los padres primerizos al incitarlos al alcoholismo!). Mejor pensado, tal vez sí que le interese, aunque sea sólo por la publicidad gratuita que le generaría. En cualquier caso, siempre podría realizar este tipo de análisis para decidir en qué pasillos y estanterías colocar cada producto, de forma que sus clientes, cuando van a comprar un producto, puedan ver el otro y acordarse de que pueden aprovechar el viaje y llevárselo también. La próxima vez que visite su supermercado, compruebe si las cervezas están colocadas sospechosamente cerca de los pañales...

En la práctica, el proceso utilizado para extraer reglas de asociación consiste en identificar todas las reglas que tengan un soporte y una confianza por encima de unos umbrales establecidos a priori por el usuario. Si no se estableciesen esos umbrales, el problema combinatorio resultante sería computacionalmente intratable.

Para extraer reglas de asociación, en primer lugar, se buscan todos los itemsets frecuentes que podrían aparecer en esas reglas. Los itemsets frecuentes son aquéllos con un soporte superior al umbral de soporte fijado por el usuario. A continuación, se generan las reglas que se derivan de los itemsets frecuentes y nos quedamos sólo con aquéllas de mayor confianza. Si $X \cup Y$ y X son itemsets frecuentes, la regla $X \Rightarrow Y$ se verifica si el cociente entre el soporte de $X \cup Y$ y el soporte de X supera el umbral de confianza mínima (el soporte de la regla es el soporte de $X \cup Y$, que ya sabemos que es frecuente).

El descubrimiento de los itemsets frecuentes es la parte del proceso de extracción de reglas de asociación más costosa computacionalmente. Por este motivo, los algoritmos de extracción de reglas de asociación se diseñan para enumerar eficientemente todos los itemsets frecuentes de una base de datos. Si un patrón dado es frecuente, sabemos que cualquier subconjunto suyo también lo será. Esta propiedad de monotonía, llamada propiedad *Apriori*, es la que nos permite descubrir patrones de forma eficiente.

Los patrones frecuentes, denominados *frequent*, *covering* o *large itemsets* en la literatura del tema, pueden identificarse construyendo un conjunto candidato de itemsets potencialmente frecuentes, que luego se comprueba si realmente lo son.

Los itemsets de tamaño k se denominan k -itemsets. Los k -itemsets frecuentes forman el conjunto $L[k]$, mientras que utilizaremos un segundo conjunto $C[k]$, en el que incluiremos aquellos k -itemsets que podrían ser

frecuentes (los candidatos).

Los algoritmos de la familia *Apriori* realizan múltiples recorridos secuenciales sobre la base de datos para obtener los conjuntos de itemsets frecuentes. En una primera pasada, se obtienen los ítems individuales cuyo soporte alcanza el umbral mínimo preestablecido para el soporte, lo que nos permite construir el conjunto $L[1]$ de ítems frecuentes. En cada iteración, utilizaremos el último conjunto $L[k]$ de k -itemsets frecuentes para generar un conjunto de $(k + 1)$ -itemsets potencialmente frecuente: el conjunto de itemsets candidatos $C[k + 1]$. Tras obtener el soporte de estos candidatos, lo que se puede conseguir con un simple recorrido secuencial de la base de datos, nos quedaremos sólo con aquéllos candidatos que son realmente frecuentes: el conjunto $L[k + 1]$. Este proceso se puede repetir iterativamente para encontrar itemsets frecuentes de un tamaño cada vez mayor.

¿Cómo se generan los candidatos? Los k -itemsets candidatos del conjunto $C[k]$ derivan directamente del conjunto de $(k - 1)$ -itemsets frecuentes $L[k - 1]$. El algoritmo original de IBM, *Apriori*, realiza la generación del conjunto de candidatos $C[k]$ a partir del conjunto $L[k - 1]$: se generan los posibles candidatos a partir del producto cartesiano $L[k - 1] \times L[k - 1]$, imponiendo la restricción de que los $k - 2$ primeros ítems de los elementos de $L[k - 1]$ coincidan (para evitar duplicados). Acto seguido, se eliminan del conjunto de candidatos aquellos itemsets que contienen algún $(k - 1)$ -itemset que no se encuentre en $L[k - 1]$. Por la propiedad Apriori, se pueden eliminar directamente de $C[k]$ aquellos itemsets que incluyan algún itemset no frecuente, puesto que sabemos que nunca podrán ser frecuentes.

La identificación de patrones frecuentes [*motif discovery*] es una operación muy habitual en minería de datos. Dado que el número de patrones puede crecer exponencialmente conforme permitimos que aumente su tamaño, se han propuesto infinidad de técnicas optimizadas para situaciones concretas, como la identificación de patrones secuenciales para el análisis de secuencias (p.ej. musicales⁴¹) o subestructuras frecuentes en estructuras de datos no lineales (p.ej. árboles⁴²).

Los patrones identificados pueden utilizarse, en un contexto no supervisado, para extraer reglas de asociación o para realizar *clustering* en subespacios, como comentamos anteriormente. Pero también pueden aprovecharse para resolver problemas de aprendizaje supervisado. Es el caso, por ejemplo, de los clasificadores asociativos.⁴³

¿Otras formas de aprendizaje?

El cerebro humano tiene del orden de 10^{11} neuronas, cada una de ellas conectada a otras muchas neuronas por medio de sinapsis. Se estima que el cerebro puede tener unas 10^{14} sinapsis (prescindamos momentáneamente

⁴¹ Aída Jiménez, Miguel Molina-Solana, Fernando Berzal, y Waldo Fajardo. Mining transposed motifs in music. *Journal of Intelligent Information Systems*, 36(1):99–115, 2011. DOI: 10.1007/s10844-010-0122-7

⁴² Aída Jiménez, Fernando Berzal, y Juan Carlos Cubero. Using trees to mine multirelational databases. *Data Mining and Knowledge Discovery*, 24(1):1–39, 2012. DOI: 10.1007/s10618-011-0218-x

⁴³ Fernando Berzal, Juan Carlos Cubero, Daniel Sánchez, y José María Serrano. ART: A Hybrid Classification Model. *Machine Learning*, 54(1):67–92, 2004. ISSN 1573-0565. DOI: 10.1023/B:MACH.0000008085.22487.a6

de la notación científica para apreciar su magnitud: ¡ $100,000,000,000,000$ sinapsis!). Sin embargo, sólo permanecemos despiertos del orden de 10^9 segundos a lo largo de nuestra vida. Si asumimos que cada sinapsis es un parámetro que ajustar en nuestro modelo mental, tendríamos que ser capaces de ajustar 100,000 parámetros por segundo (10^{14} sinapsis / 10^9 segundos). Según Geoffrey Hinton, de la Universidad de Toronto, esto sugiere que gran parte de nuestro aprendizaje ha de ser no supervisado. Nuestra percepción, incluyendo la propiocepción, es la única fuente de la que podríamos obtener información suficiente para ajustar tantísimos parámetros.

Tal vez el aprendizaje no supervisado sea esencial, aunque no hay que descartar otros tipos de aprendizaje supervisado en los que la supervisión nos llega por vías alternativas, a menudo indirectas. De hecho, si le preguntamos a muchos investigadores en Inteligencia Artificial por los distintos tipos de aprendizaje, al aprendizaje supervisado y no supervisado que ya hemos visto, le añadirían un tercero: el aprendizaje por refuerzo. ¿De dónde proviene ese tercer tipo de aprendizaje? De una de las formas mediante las que aprendemos los animales, algunos humanos incluidos.

El proceso de aprendizaje animal incluye cualquier mecanismo que nos permita modificar nuestro comportamiento en el futuro, una forma de proyectar la información de la que disponemos en el presente para predecir el futuro (y adaptar nuestro comportamiento a tal predicción). Por eso un evento inesperado nos llama mucho la atención y algo que dábamos por sentado ocurre sin que apenas nos demos cuenta.

¿Cómo aprenden los animales? Básicamente, de tres formas diferentes:⁴⁴ por refuerzo, por asociación y por imitación.

Aprendizaje por refuerzo

El aprendizaje por refuerzo es el que se consigue cuando felicitamos a nuestro perro con una palmadita cada vez que hace lo que nosotros queremos. También se le conoce por los términos condicionamiento operante y condicionamiento instrumental.

El psicólogo estadounidense B.F. Skinner utilizó el aprendizaje por refuerzo en sus experimentos con palomas, un proyecto que fue cancelado antes de ser operativo, por motivos obvios. En ese proyecto, Skinner pretendía entrenar palomas para usarlas en la II Guerra Mundial como proyectiles suicidas. Skinner las condicionaba para que siguieran y picotearan una figura determinada en busca de alimento. A continuación, mediante una placa transparente en la que se reflejase la figura de un objetivo, como un barco, un avión o un tanque, la paloma orientaría su cuerpo hacia el objetivo. Encerrando a la paloma en un dispositivo que aprovechase sus movimientos para corregir su rumbo, la paloma podría guiarlo hasta el objetivo, aunque éste se encontrase en movimiento. Evi-

En realidad, es más que posible que los 10^{14} parámetros se estén ajustando continuamente, durante los más de 10^9 segundos de nuestra vida, tanto en el sueño como durante la vigilia.

Consulta rápida a Wikipedia: La propiocepción es el sentido que informa al organismo de la posición de los músculos (un complemento necesario a la percepción de estímulos sensoriales a través de los sentidos para que podamos desenvolvernos).

⁴⁴ Maureen Caudill y Charles Butler. *Naturally Intelligent Systems*. MIT Press, 1st edition, 1990. ISBN 0262031566

dentemente, en esa época no se disponía de drones ni de misiles guiados por satélite.

El aprendizaje instrumental es la base de cualquier manual de entrenamiento para perros, la forma en que podemos conseguir que el perro se comporte como queremos. Si queremos que se siente, por ejemplo, se lo decimos verbalmente o se lo indicamos con la mano. Cuando el perro se sienta, lo recompensamos inmediatamente, para que establezca una asociación entre su acto y la recompensa. La señal, verbal o visual, sirve de estímulo para el perro y su acción es la respuesta que deseamos obtener cada vez que le proporcionemos el mismo estímulo. Esta forma de aprendizaje se denomina condicionamiento operante porque así enseñamos al animal a operar en su entorno.

El psicólogo estadounidense Edward Thorndike describió varias leyes que gobiernan el aprendizaje por refuerzo. Su ley del efecto implica que las relaciones estímulo-respuesta [S-R] dependen de lo mucho que al animal le guste su recompensa. Cuanto más le guste, mayor será la fortaleza del enlace creado entre estímulo y respuesta. Su ley del ejercicio establece que una relación estímulo-respuesta se fortalece con el uso y se debilita con el desuso (como casi cualquier habilidad en la práctica, por otro lado). De hecho, Skinner llegó a pensar que todo el comportamiento puede reducirse a un conjunto de relaciones estímulo-respuesta.

Para que un cerebro, animal o humano, sea eficiente, debe ser capaz de aprender, obviamente. No obstante, para que el aprendizaje en sí sea eficiente, se ha observado que resulta aconsejable que exista cierta incertidumbre en la consecución de recompensas. El aprendizaje resulta más efectivo si las recompensas son episódicas e impredecibles. ¿Por qué? Porque no recompensar todo intento puede servir para incrementar nuestra motivación (y la de nuestro perro). Esta estrategia se conoce con el nombre de refuerzo variable [VR: Variable Reinforcement]. El refuerzo variable es muy común en el entrenamiento animal: una estrategia VR10, por ejemplo, indica que el sujeto recibe una recompensa ocasionalmente pero, en media, sólo en una de cada diez pruebas. La impredecibilidad del refuerzo variable es lo que consigue que los animales muestren mayor atención y se esfuerzen más en conseguir su recompensa.

Volvamos al aprendizaje automático. Mientras que en las técnicas supervisadas se trabaja con datos etiquetados y en las no supervisadas es el propio algoritmo el que reconoce patrones y etiqueta los datos, el aprendizaje por refuerzo consiste, básicamente, en un mecanismo de prueba y error. Tras un ensayo con éxito, la estrategia utilizada es recompensada (refuerzo positivo). Tras un fallo, la estrategia que nos llevó a equivocarnos es penalizada (refuerzo negativo). La realimentación positiva recibida por realizar las acciones que nos condujeron a lograr un objetivo concreto es la que sirve de señal de supervisión en el aprendizaje por refuerzo. Por tanto, éste puede interpretarse como una forma más de

aprendizaje supervisado.

La forma más eficiente de aprender por refuerzo, ya sea una máquina o un animal, es utilizar un método de diferencias temporales: TD-learning [temporal difference learning]. La idea que subyace a este método es ir ajustando nuestras predicciones para que encajen mejor con el futuro. Se usa como señal de error la diferencia entre la recompensa esperada y la recompensa real observada. Si la recompensa real es superior a la esperada, incentivamos nuestro comportamiento aumentando nuestras expectativas. Si es inferior, intentamos rebajar nuestras expectativas para que, en el futuro, nuestra predicción se acerque más al resultado observado.

Aprendizaje por asociación

El aprendizaje por asociación corresponde al condicionamiento clásico, el de los experimentos de los perros de Pavlov de los que seguramente habrá oído hablar. De hecho, algunos siguen llamándole condicionamiento Pavloviano en su honor.

A diferencia de los que se preocupan por el correcto adiestramiento de perros, el psicólogo ruso Ivan Pavlov no es que apreciase demasiado a estos fieles compañeros del hombre, lobos domesticados desde el punto de vista genético. Pavlov se limitaba a intentar estudiar su aparato digestivo. El problema es que los perros comenzaban a salivar antes de recibir su comida, lo que le estropeaba los datos de su estudio a Pavlov. Su experimento ‘fallido’, no obstante, condujo al descubrimiento más importante de la Psicología en el siglo XX, por lo que recibió el premio Nobel en 1904.

En esa época, los fisiólogos pensaban que el sistema nervioso completo no era más que una colección de reflejos, como la contracción involuntaria del cuádriceps de su pierna cuando recibe un pequeño golpe en el tendón rotuliano o patellar de su rodilla. Antes de Skinner, que reducía el comportamiento a relaciones estímulo-respuesta, la situación era aún peor: todo el comportamiento, por complejo que fuese, se interpretaba como una serie de reflejos. Un reflejo se descomponía en un estímulo no condicionado [US: unconditioned stimulus] y una respuesta no condicionada [UR: unconditioned response].

Pavlov observó que sus perros mostraban un comportamiento, para él, poco natural. Los perros hambrientos siempre salivan cuando ven comida, es una respuesta natural no condicionada. Sin embargo, en ocasiones, un estímulo inicialmente neutral, como el sonido de un timbre que precede a la llegada de la comida, terminará evocando la respuesta no condicionada con su sola presencia. El timbre se convierte en un estímulo condicionado [CS: conditioned stimulus] que evoca una respuesta condicionada [CR: conditioned response]: la salivación. El término condicionado se utiliza

aquí para hacer referencia a las condiciones creadas por el experimentador. Estímulos y respuestas no condicionados son los ya presentes sin intervención del experimentador.

En los años 90, el neurocientífico suizo Wolfram Schultz realizó otro experimento de condicionamiento clásico, esta vez con monos. Cuando se encendía una luz, el mono recibía un chorro de zumo de fruta en su boca. Como los perros de Pavlov, el mono rápidamente comenzó a asociar el zumo a la luz. Midiendo la actividad de su cerebro, descubrió qué neuronas en zonas específicas del cerebro seguían el mismo patrón. Inicialmente, se activaban como respuesta al zumo. Tras aprender la asociación de la luz con el zumo, esas neuronas se activaban con la presencia de la luz, sin zumo. Esas neuronas se encuentran en la zona del cerebro que se asocia a su sistema de recompensas: el núcleo caudado, parte de los ganglios basales. Esas neuronas no se activan con experiencias placenteras, sino cuando ocurre algo inesperado que crea la expectativa de que suceda algo placentero. Cuando ocurre algo inesperado, el cerebro detecta un error en su predicción, por lo que esos eventos se denominan errores en la predicción de recompensas.

El condicionamiento clásico, obviamente, no sirve de mucho si lo que quiere es adiestrar a su perro. Las respuestas que incita son tan simples que no se corresponden con un comportamiento útil, aquél que tenga un objetivo concreto con significado. Si quiere enseñar a su perro a dar la pata, no utilice el condicionamiento Pavloviano, recurra al condicionamiento operante. El condicionamiento clásico sólo genera una respuesta involuntaria (como la salivación), mientras que el condicionamiento operante sí puede servir para entrenar un comportamiento voluntario.

No obstante, sí que se puede utilizar para modificar su comportamiento. En el aprendizaje por refuerzo con *TD-learning*, se compara la recompensa esperada con la recompensa observada, lo que sirve como señal de error. Podemos aprovechar esa señal de error para asociar una futura recompensa con un estímulo que anticipe fielmente una futura recompensa. Cuando un animal aprende que ese estímulo, visual o acústico, predice una recompensa de forma precisa, disminuye su nivel de activación ante la presencia de la recompensa en sí y aumenta para la señal que la anticipa. Algo así puede servirnos para que el perro no nos exija comida, le pierda el miedo a las visitas al veterinario o suprima algún trauma infantil (mi perra le tiene pánico a los niños pequeños después de que un grupo grande de ellos corriese hacia ella cuando era aún un cachorro). Mediante aprendizaje condicionado, se pueden cambiar algunas asociaciones que inducen un comportamiento no deseado en nuestras mascotas.

Aprendizaje por imitación

El aprendizaje por imitación también se conoce como aprendizaje social o vicario. También podríamos llamarlo aprendizaje por observación. Fundamental para el aprendizaje humano, también se puede observar claramente en animales. Se conoce relativamente poco acerca de la neurobiología asociada al aprendizaje social o por imitación, ni siquiera acerca de las partes del cerebro involucradas.

En los animales, se solía pensar que el aprendizaje asociativo dominaba, creando relaciones entre eventos y cosas que nos gustan o nos disgustan. Pero no resulta suficiente. Por un motivo muy sencillo: es extremadamente ineficiente. Por eso leemos libros, asistimos a cursos y seminarios o pasamos gran parte de nuestra vida infantil en un entorno escolar. El aprendizaje asociativo requiere experimentar las asociaciones, un proceso de prueba y error. Tendríamos que quemarnos físicamente con fuego para aprender que con fuego no se juega.

El aprendizaje social es mucho más eficiente. No sólo entre nosotros, muchos animales también lo emplean. Algunos pájaros aprenden por imitación sus cantares característicos (o a insultar a la suegra si es un loro mal educado, especialmente cuando viene de visita). Los perros también aprenden unos de otros desde cachorros, imitando a su madre y a los miembros de su camada. Incluso es posible que acabasen domesticados gracias a su gran capacidad de empatía. Los lobos más observadores supieron ver algo positivo en los primeros asentamientos humanos y se quedaron con nosotros desde entonces, cuidando rebaños, ayudando en la caza o avisando ante la presencia de intrusos.

Los perros, además de ser capaces de identificar hacia dónde dirigimos nuestra atención, son capaces de distinguir situaciones dependiendo del contexto. Saben cuándo deben prestar atención y cuándo pueden volver a su cojín. Eso requiere algo más que un simple comportamiento reactivo. Requiere una teoría de la mente [ToM: Theory of Mind].⁴⁵ En psicología, una teoría de la mente designa la capacidad de imaginar lo que otro está pensando: atribuir pensamientos e intenciones a otras personas (o animales). Dada la complejidad de nuestra vida social, se cree que gran parte de nuestros lóbulos frontales se dedican a ello. Saber cómo interpretar a otras personas y cómo comportarse en diferentes situaciones puede ser la diferencia entre el éxito y el fracaso. Determinadas enfermedades, como casos graves de autismo, pueden estar relacionadas con fallos en el sistema ToM del cerebro. Obviamente, aunque los perros tengan capacidades ToM, las suyas son más simples que las nuestras (entre otras cosas, por el tamaño del lóbulo frontal de su cerebro). Pero aunque sean capacidades rudimentarias, son indicativas de que los perros no son simples máquinas estímulo-respuesta. Puede que indiquen un grado de conciencia similar al de un bebé.

⁴⁵ Gregory Berns. *How Dogs Love Us: A Neuroscientist and His Adopted Dog Decode the Canine Brain*. New Harvest, 2013. ISBN 0544114515

¿A que no esperaba encontrar una referencia a un estudio neurocientífico sobre psicología perruna en un libro de redes neuronales artificiales?

Un conductista recalcitrante, tipo Skinner, diría que los perros se limitan a establecer asociaciones entre estímulos neutrales y respuestas no condicionadas. Para ellos, nada en el cerebro implica que se entienda el “significado” de algo. Sin embargo, un perro es capaz de interpretar los matices de un simple gesto con la mano y otras señales sociales utilizadas por los humanos, como apuntar a algo con el dedo y que el animal sepa rápidamente a dónde tiene que dirigirse. Lográndolo a la primera. Una capacidad que un conductista es incapaz de explicar. Curiosamente, es algo que los lobos y los chimpancés no consiguen hacer tan bien como los perros (aun siendo los lobos genéticamente indistinguibles de los perros).

Según algunos antropólogos evolutivos, como Brian Hare, de la Universidad de Duke, el aprendizaje social es la clave que marcó nuestro desarrollo como especie. Lo que nos hace verdaderamente distintos a otros animales. Los bebés desarrollan habilidades sociales con apenas nueve meses, tras lo que comienzan a imitar el comportamiento de otros y empiezan a pronunciar sus primeras palabras. Esas habilidades sociales son las que permiten a los peques de la casa comunicarse con los adultos con los que comparten comida, refugio, juegos y objetos de todo tipo, lo que a su vez alimenta su deseo de cooperación. La posibilidad de leer las intenciones de otros, gracias a nuestras capacidades ToM, es la base de todas las formas humanas de cultura y comunicación, tanto del lenguaje (la más compleja) como de los gestos visuales. Sin ellas, no podríamos progresar aprovechando los logros de las generaciones que nos precedieron.

Su uso en aprendizaje automático

Tras los descubrimientos sobre aprendizaje animal de Pavlov, Thorndike y Skinner, se desarrolló la psicología del comportamiento, muy popular en los años 60. Los psicólogos conductistas han aplicado las distintas teorías de aprendizaje animal al comportamiento humano, desde dejar de fumar hasta aprender a hacer nuevos amigos.

Ahora bien, ¿se utilizan en Inteligencia Artificial las técnicas basadas en las teorías de aprendizaje animal? ¿Aprenden los ordenadores por refuerzo, por asociación y por imitación? Sí, no y tal vez. El aprendizaje por refuerzo suele ser útil para diseñar sistemas en los que la señal de supervisión no es inmediata. El aprendizaje por asociación, que podemos utilizar para corregir comportamientos no deseados en nuestro perro, no se suele emplear en aprendizaje automático. El aprendizaje por imitación, aunque no se suela emplear actualmente, abre la posibilidad de mejorar las capacidades de aprendizaje de los sistemas futuros.

El aprendizaje por refuerzo sí se ha utilizado con cierta frecuencia en sistemas de aprendizaje automático desde hace décadas.⁴⁶ Más re-

Viviendo entre un grupo de animales, es importante predecir lo que otros van a hacer y los gestos visuales proporcionan información social que nos permiten adivinar las intenciones de otros. La supervivencia de un lobo solitario, a diferencia de un perro, que suele ser gregario, no depende tanto de la interpretación de esos gestos visuales. Los perros no son más que descendientes de los lobos de hace miles de años. En particular de aquellos que se auto-seleccionaron al mostrar una mayor inclinación a colaborar socialmente, incluso con el hombre. Algo similar se ha observado en zorros ‘domesticados’ en Siberia o en comunidades de bonobos, mucho más cooperativos que los agresivos chimpancés.

Estas ideas se siguen utilizando en las terapias cognitivo-conductuales [CBT: Cognitive-Behavioral Therapy] para tratar trastornos como la depresión o la ansiedad.

⁴⁶ Richard S. Sutton y Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1st edition, 1998. ISBN 0262193981

cientemente, se ha incorporado al conjunto de técnicas empleadas para entrenar redes neuronales cuando la señal de supervisión que le permite al sistema ajustar su comportamiento no es inmediata:

- *Aprendizaje por refuerzo en juegos*

Existen algunos sistemas basados en redes neuronales que utilizan aprendizaje por refuerzo para distintos tipos de juegos. Una de las primeras demostraciones de DeepMind fue un sistema que aprendía solo a jugar a videojuegos de Atari.⁴⁷ Al año siguiente, AlphaGo fue capaz de vencer a campeones humanos de Go.⁴⁸ Y también se han desarrollado jugadores artificiales capaces de jugar al póker como un jugador profesional (DeepStack).⁴⁹

Según Demis Hassabis, líder del equipo de DeepMind tras el desarrollo de AlphaGo, la máquina consigue algo cercano a imitar la intuición humana combinando técnicas de aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo (que no deja de ser una forma de aprendizaje supervisado con una señal de supervisión en diferido).

- *Aprendizaje por refuerzo en procesamiento del lenguaje natural*

Richard Socher, fundador de MetaMind antes de que fuese adquirida por Salesforce en 2016, también ha combinado redes neuronales con aprendizaje por refuerzo, pero esta vez para resumir textos. Tradicionalmente, se han empleado dos enfoques diferentes para resumir textos. El primero de ellos consiste en seleccionar pasajes del texto original y utilizarlos directamente para redactar el resumen. Son los denominados resúmenes extractivos. El segundo enfoque pretende abstraer el contenido del texto y elaborar el resumen redactando para ello un texto nuevo, formado por frases que no tienen por qué aparecer en el texto original. Son los resúmenes abstractivos.

La forma en que decimos (o escribimos) algo depende del contexto de lo que hemos dicho (o escrito) antes, por lo que, dotando a su sistema de un mecanismo de atención, Socher consigue imitar la forma mediante la que nosotros prestamos atención a lo que acabamos de leer o de decir. Esto se traduce en la obtención de resúmenes abstractivos más legibles.⁵⁰

El aprendizaje por imitación, sin embargo, es algo casi anecdotico en los sistemas actuales de aprendizaje automático. Sólo se ha probado con éxito en entornos muy limitados. Algunos robots consiguen imitar las tareas que realizan otros robots. Básicamente, se limitan a reproducir de forma precisa la secuencia de operaciones que previamente hayan sido capaces de observar. Nada que se acerque, ni de lejos, al nivel de inteligencia (y empatía) ofrecido por un animal de compañía, por muy colejos que resultasen robots como AIBO, diseñado por Sony a finales del siglo pasado.

⁴⁷ Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, y Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. DOI: 10.1038/nature14236

⁴⁸ David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, y Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. DOI: 10.1038/nature16961

⁴⁹ Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, y Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017. ISSN 0036-8075. DOI: 10.1126/science.aam6960

⁵⁰ Romain Paulus, Caiming Xiong, y Richard Socher. A deep reinforced model for abstractive summarization. *arXiv e-prints*, arXiv:1705.04304, 2017. URL <http://arxiv.org/abs/1705.04304>

Evaluación de los resultados del aprendizaje

Sea cual sea el tipo de técnica de aprendizaje que decidamos emplear para resolver los problemas a los que nos enfrentemos, todas ellas tienen algo en común. El resultado del proceso de aprendizaje será un modelo. Ese modelo, como todos los modelos, no es más que una simplificación de la realidad. Y, al ser una simplificación, estará sujeto a errores. Por tanto, deberíamos evaluar la calidad del modelo obtenido para ver si realmente nos puede servir en la práctica.

Pensemos en lo que ocurre, en la vida real, cuando queremos resolver un problema. Si no estamos familiarizados con el problema en cuestión, podríamos acudir a expertos humanos sobre el tema. Esos expertos, dada su experiencia previa, son capaces de realizar un diagnóstico concreto para una situación dada, generalmente acertado (o eso esperamos, salvo que se trate de economistas). De hecho, la construcción de muchos sistemas basados en el conocimiento, como pueden ser los sistemas expertos de diagnóstico, se realiza entrevistando a expertos en el tema. Sin embargo, la extracción manual del conocimiento de los expertos entraña bastantes dificultades. Los expertos no siempre son capaces de articular el motivo de sus decisiones. ¿Por qué? Porque se basan en conocimiento implícito que tienen sobre el problema, difícil de describir con precisión de forma explícita. La paradoja de la Ingeniería del Conocimiento es que, cuanto mejor es el experto, peor suele describir su conocimiento. A todo esto hay que añadirle el hecho de que los expertos no siempre están de acuerdo y, aun cuando lo están, puede que no lo estén por los mismos motivos. Como corolario de todo esto se deriva la Ley de Hiram: “Si se consultan suficientes expertos, se puede confirmar cualquier opinión”.

Otra alternativa que nos ofrecen las técnicas de aprendizaje en I.A. es poder construir un modelo de forma automática. En vez de intentar construir un modelo de forma manual consultando expertos, lo que haremos será recopilar datos que recojan las variables que creamos que pueden ser relevantes para nuestro problema. Los datos utilizados en la construcción del modelo describen casos particulares del problema que pretendemos modelar. Esos casos, denominados ejemplos en aprendizaje automático o patrones en reconocimiento de formas, se describen, usualmente, en términos de un conjunto finito de variables, propiedades o atributos. En un problema de aprendizaje supervisado, además, cada caso de entrenamiento irá etiquetado con el valor de salida esperado que se asigna al

Se suele decir que Winston Churchill, cuando pedía consejo a sus cinco economistas de cabecera, siempre obtenía seis opiniones diferentes (uno de sus asesores era Keynes).

ejemplo concreto en el conjunto de entrenamiento (una clase o categoría en los problemas de clasificación, un valor numérico en los problemas de regresión).

Una vez que dispongamos de suficientes datos, utilizaremos esos datos para construir un modelo mediante un proceso de entrenamiento. Ese entrenamiento será el que le permita al modelo ser capaz de generalizar a partir de ejemplos concretos. El entrenamiento dará como resultado un modelo aplicable a datos diferentes de los utilizados en su entrenamiento. ¿Cuántos datos serán necesarios? Para que el aprendizaje automático sea correcto, entendido éste como un proceso de generalización a partir de ejemplos concretos, hemos de disponer de suficientes casos de entrenamiento (bastantes más que clases diferentes en un problema de clasificación, por ejemplo). Si las conclusiones derivadas del modelo aprendido no vienen avaladas por bastantes ejemplos, entonces la inevitable existencia de errores y ruido en los datos de entrenamiento nos podría conducir al aprendizaje de un modelo erróneo, el cual no resultaría fiable en la práctica. Cuantos más datos obtengamos, más fácilmente podremos diferenciar patrones válidos de patrones debidos a irregularidades o errores en el conjunto de entrenamiento. En minería de datos o *big data*, cuantos más datos usemos, mejores pueden llegar a ser los resultados obtenidos.

Ahora bien, una vez construido un modelo a partir de un conjunto de datos de entrenamiento, nos gustaría poder evaluar su calidad, para saber cómo de bueno sería en la práctica si decidimos utilizarlo en el mundo real. Para ello, diferenciaremos dos aspectos complementarios:

- Las métricas de evaluación: Cómo medir la calidad de un modelo obtenido mediante el uso de técnicas de aprendizaje automático.
- Los métodos de evaluación: Cómo realizar una estimación fiable de las métricas que nos indican la calidad de un modelo.

Métricas de evaluación

Comencemos por el caso más habitual y sencillo: un problema de clasificación binaria. Deseamos construir un modelo de clasificación que nos permita discriminar entre dos clases diferentes. A la primera de ellas, que corresponde a los ejemplos que deseamos ser capaces de identificar, la denominaremos clase positiva (P). Los demás ejemplos los asignaremos a la clase negativa (N). Cuando aplicamos un modelo de clasificación a un conjunto de datos previamente etiquetado, podemos ver fácilmente cómo etiqueta los diferentes ejemplos nuestro clasificador. Sólo existen cuatro posibilidades, que podemos representar en una matriz de contingencia, también llamada matriz de confusión:

- Verdaderos positivos [*TP: True Positive*]: Los ejemplos de la clase positiva que nuestro clasificador es capaz de clasificar correctamente.
- Falsos positivos [*FP: False Positive*]: Los ejemplos que, aun no siendo de la clase positiva, nuestro clasificador predice que sí lo son.
- Falsos negativos [*FN: False Negative*]: Los errores que comete nuestro clasificador en sentido contrario, diciéndonos que no son de la clase positiva cuando en realidad sí que lo son.
- Verdaderos negativos [*TN: True Negative*]: Los ejemplos de la clase negativa que nuestro clasificador clasifica correctamente.

Los cuatro casos posibles se recogen en una sencilla matriz de confusión de tamaño 2×2 en el caso de los problemas de clasificación binaria. Si nos encontramos ante un problema de clasificación con múltiples clases, la matriz de confusión tendría una dimensión de $K \times K$, donde K es el número de clases del problema. Analizando las filas y columnas de esa matriz podemos ver dónde acierta y dónde se equivoca nuestro modelo de clasificación, lo que nos puede indicar posibles problemas y sugerir qué es lo que deberíamos intentar cambiar.

Ahora bien, aparte de poder analizar con detalle cómo clasifica nuestro modelo de clasificación los ejemplos de las diferentes clases, nos gustaría poder disponer de medidas numéricas que nos ayudasen a resumir el contenido de la matriz de contingencia y comparar modelos de clasificación alternativos.

Precisión y tasa de error

La métrica más utilizada para resumir el rendimiento de un modelo de aprendizaje supervisado es, sin duda, su precisión [*accuracy*]. La precisión nos indica la proporción de ejemplos que un clasificador es capaz de clasificar correctamente, indicada habitualmente en forma de tanto por ciento:

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

Para abreviar, llamaremos n al número de ejemplos del conjunto de datos sobre el que estemos evaluando nuestro modelo. En un problema de clasificación binaria, $n = TP + TN + FP + FN$, por lo que

$$\text{precisión} = \text{accuracy} = \frac{TP + TN}{n} = \frac{\text{aciertos}}{n}$$

Si tenemos un problema con k clases diferentes, simplemente sumaríamos los elementos correspondientes a la diagonal de la matriz de confusión, que corresponden a los aciertos de nuestro clasificador para cada una de las clases de nuestro problema.

		Predicción	
		P	N
Clase real	P	TP	FN
	N	FP	TN

Figura 5: Matriz de confusión para un problema de clasificación binaria.

En los problemas de clasificación con K clases diferentes es habitual, en muchas ocasiones, construir K clasificadores binarios diferentes, uno para cada clase. Esta estrategia, denominada *1 vs. all*, puede ser útil si utilizamos técnicas de aprendizaje supervisado que no están diseñadas para discriminar entre K clases diferentes simultáneamente.

		Predicción	
		P	N
Clase real	P	TP	FN
	N	FP	TN

Figura 6: La precisión [*accuracy*] de un clasificador nos indica el número de aciertos que consigue a la hora de clasificar ejemplos.

		Predicción	
		P	N
Clase real	P	TP	FN
	N	FP	TN

Figura 7: La tasa de error de un clasificador contabiliza los errores que comete.

De forma alternativa, podríamos contabilizar los fallos de nuestro clasificador y resumir su rendimiento mediante una simple tasa de error:

$$\text{error} = \frac{FP + FN}{n} = \frac{\text{errores}}{n}$$

En algunas situaciones, se hace referencia a la tasa de error como pérdida 0-1 esperada [*expected 0-1 loss*]. La pérdida 0-1 para un ejemplo particular es 0 cuando éste se clasifica correctamente, 1 cuando no. Su esperanza, por tanto, es la tasa de error del clasificador.

Evidentemente, da igual que indiquemos precisión [*accuracy*] o tasa de error, puesto que se trata de medidas complementarias:

$$\text{precisión} = \text{accuracy} = 1 - \text{error}$$

Ingenuamente, podríamos pensar que nuestro objetivo será conseguir una tasa de acierto del 100 %, lo que equivale a reducir el error al cero absoluto. Sin embargo, generalmente tendremos que conformarnos con algo mucho más modesto. Nunca conseguiremos eliminar por completo el error en nuestros modelos (entre otras cosas, porque son modelos).

¿Cómo sabemos si nuestro clasificador es lo suficientemente bueno? Para responder a esta pregunta, nos puede venir bien disponer de algún punto de referencia con el que establecer una comparación. En ocasiones, nos bastará con utilizar un modelo de clasificación simple y ver hasta qué punto nuestro modelo mejora los resultados del clasificador base. En términos relativos, podemos ofrecer nuestros resultados en forma de mejora [*lift*]. Si un clasificador base nos proporciona una precisión del 75 % y nuestro clasificador aumenta esa cifra hasta el 90 %, podemos decir que nuestro clasificador proporciona una mejora del 15 % en su precisión, un *lift* del 20 % ($(15\% / 75\%)$), una reducción del error del 15 % o una reducción de la tasa de error del 60 % ($((25\% - 10\%) / 25\%)$). Para vender nuestro clasificador, posiblemente nos quedaríamos con la última opción, que resulta mucho más llamativa.

¿Existe la posibilidad de mejorar el rendimiento de nuestro clasificador? En este caso, también nos sería de utilidad encontrar un punto de referencia que nos sirva de objetivo. Al realizar el mismo trabajo que nuestro clasificador, los seres humanos no son perfectos y suelen cometer errores. Esta tasa de error puede deberse a las dificultades y ambigüedades inherentes al problema de clasificación. No tiene por qué estar causada por la dejadez e incompetencia de los humanos al realizar esa tarea. Si podemos conseguir una estimación de la tasa de error cometida por el ser humano en el problema que pretendemos automatizar, nos puede servir para fijar un sistema de referencia. A este valor se le suele denominar “patrón oro” o *gold standard* en inglés, en referencia al régimen monetario en el cual se podía intercambiar una cantidad de dinero por su valor en oro, que se mantenía fijo (35 dólares por onza de oro desde

1933 hasta 1971). En ocasiones se utiliza este valor para determinar cuándo un problema ha sido resuelto en Inteligencia Artificial. Hasta hace muy poco, los sistemas de reconocimiento de voz eran peores que los seres humanos. Algo que cambió gracias al uso de redes neuronales y que posibilitó el despegue de los asistentes virtuales de los que dispone en su smartphone.

Medidas sensibles a costes

No siempre resulta adecuado utilizar una simple tasa de acierto o error para caracterizar la calidad de un modelo. Por ejemplo, hay problemas en los que no es lo mismo un falso positivo (“¡cuidado! esta seta es venenosa y no debería comérsela”) que un falso negativo (“coma tranquilo, esta seta no es venenosa”... pero en realidad sí que lo era). En el ejemplo de la clasificación de setas, la diferencia puede ser una cuestión de vida o muerte. Un falso positivo le privaría de una experiencia gastronómica. Un falso negativo le haría necesitar un trasplante de hígado. De ahí que se deban extremar las precauciones para evitar falsos negativos. En otros problemas, la situación puede suceder a la inversa. Un sistema que se utilice de alarma, ya sea para la detección de fraudes en operaciones con tarjetas de crédito o la detección de intrusiones en seguridad, debería evitar un excesivo número de falsos positivos. Si éstos se producen, probablemente el operador humano deje de prestarles atención a las alertas generadas por el sistema (suponiendo que no opte, simplemente, por desconectar el sistema para que no moleste). Pueden existir situaciones en que los matices del problema puedan influir en nuestra interpretación de la calidad del modelo de clasificación construido. Por ejemplo, no es lo mismo un falso positivo en la detección de cáncer (pruebas posteriores de diagnóstico podrían descartar esa posibilidad) que un falso negativo en la detección de Ébola (dejando que alguien infectado se pasee tranquilamente por la ciudad).

Un ejemplo típico de los libros de Estadística es la paradoja de los falsos positivos. Tenemos una prueba diagnóstica que tiene una precisión del 99 % a la hora de determinar correctamente si un paciente padece o no una enfermedad no demasiado común, que supongamos que sólo afecta a una de cada mil personas, un exiguo 0.1 % de la población. ¿Cuál es la probabilidad de que tengamos realmente la enfermedad tras recibir un diagnóstico positivo?

En nuestra tabla de contingencia, construida a partir de una población de 100,000 personas, tendremos que la prueba diagnóstica clasifica correctamente 99 de los 100 casos en los que se observa la enfermedad. Esto es, $TP = 99$ y $FN = 1$. Sin embargo, la gran mayoría de la población no padece esa enfermedad. En la segunda columna de la tabla, $FP = 999$ y $TN = 98,901$ (para conservar nuestro 99 % de precisión global). Es decir, la prueba diagnóstica da positivo para un total de 1,098 casos

Actualmente, el sistema fiduciario no vincula la unidad monetaria a su peso en oro, lo que permite que los bancos centrales creen de la nada todo el dinero que estimen oportuno, con las consecuencias que eso puede ocasionar. ¿Es de los que piensa que, debajo de un colchón, su dinero está protegido? ¿que conserva su valor? Piénselo dos veces...

		Predicción	
		P	N
Clase real	P	99	1
	N	999	98901

Figura 8: La paradoja de los falsos positivos en una prueba diagnóstica con el 99 % de precisión.

(99 para los que tienen la enfermedad, TP, más 999 para los que no la tienen). Si recibimos un diagnóstico positivo, sólo en el 9 % se tratará de un diagnóstico correcto (99/1098). Aunque el diagnóstico acierte en el 99 % de los casos cuando ya tenemos la enfermedad, el diagnóstico positivo cuando no sabemos si tenemos la enfermedad sólo nos da un 9 % de probabilidad de tener la enfermedad realmente. Matemáticamente, $p(D|E) = 0.99$ pero $p(E|D) = 0.09$. Éste es el motivo por el cual su médico, si tiene que diagnosticar una enfermedad grave, debería realizar múltiples pruebas que confirmen el diagnóstico y minimicen la posibilidad de un falso positivo antes de asustarle como paciente (si no sabe probabilidades).

En situaciones como las anteriores, se pueden asociar costes a las diferentes decisiones de un clasificador. Es habitual que estos costes se presenten en términos monetarios, de forma que tendrá que estimar cuál es el impacto real de cada una de las situaciones reflejadas en su matriz de contingencia para poder comparar modelos alternativos. Si un falso positivo es mucho más costoso que un falso negativo, su matriz de costes debería reflejar este hecho. Por ejemplo, para un banco, un crédito hipotecario concedido a quien no debería concedérselo, porque terminará no pagándolo, es, en principio, más costoso que un falso negativo (un caso en el que deja de ganar los réditos que le reporta el interés de la hipoteca, descontada inflación). Asumiendo, claro está, la ausencia de riesgo moral.

Medidas de calidad como la precisión [*accuracy*] o la tasa de error tampoco resultan adecuadas en problemas de clasificación con clases poco equilibradas, los problemas de clasificación no balanceada. Supongamos que, para un problema determinado, disponemos de un conjunto de datos con 10,000 ejemplos. De esos ejemplos, sólo 10 corresponden a la clase P, mientras que los 9,990 restantes son de la clase N. En este caso, podríamos construir un clasificador por defecto, que siempre diga que los ejemplos son de la clase más frecuente. Ese clasificador, aunque totalmente inútil para resolver nuestro problema de clasificación, obtendrá una precisión del 99.9 %, difícilmente mejorable. Eso sí, la elevada precisión del clasificador es totalmente engañosa, ya que será incapaz de detectar un solo ejemplo de la clase P. Aunque pueda parecer un caso ficticio, es muy habitual en la práctica. Todos los sistemas de detección de anomalías presentan características similares, ya sean para realizar el control de calidad en una fábrica, detectar fallos en sistemas mecánicos, marcar como sospechosas operaciones financieras fraudulentas, detectar intrusiones en un sistema o identificar el comportamiento anómalo de un programa que podría indicar la presencia de virus informáticos. En todas esas aplicaciones, tendremos que afinar y no resumir la calidad de nuestro modelo con una simple medida de precisión o error.

$p(D|E)$ denota la probabilidad del diagnóstico positivo D dada la enfermedad E , mientras que $p(E|D)$ denota la probabilidad de la enfermedad E dado el diagnóstico positivo D .

El riesgo moral hace referencia a situaciones en las que un individuo tiene información acerca de las consecuencias de sus propias acciones y, sin embargo, son otras personas las que soportan las consecuencias de los riesgos asumidos (p.ej. los rescates bancarios).

Evaluación de rankings

En muchos sistemas, no nos interesa tanto una decisión binaria como establecer un orden entre las diferentes hipótesis plausibles. Es el caso de los sistemas de recuperación de información, los sistemas QA [*Question Answering*] o los sistemas de recomendación. En todos ellos, el sistema genera una lista ordenada de posibles respuestas. En algunos casos, los primeros elementos de esa lista serán los que se le muestren al usuario como respuesta y es precisamente ése el conjunto que hemos de evaluar. En otros casos, como los sistemas QA, puede que busquemos una respuesta correcta en particular y nos gustaría que ésta apareciese lo más alto posible en la lista ordenada de posibles respuestas.

Para evaluar un sistema de recuperación de información se suelen emplear dos medidas complementarias, precisión y exhaustividad (*recall* en inglés). La primera de ellas nos indicará cómo de precisos son los resultados de nuestro sistema de recuperación de información (si se incluyen demasiados errores en los resultados de una búsqueda), mientras que la segunda de ellas nos indicará cómo de exhaustivos son los resultados obtenidos (si incluyen todos los resultados que deberían incluirse).

La medida de precisión utilizada en sistemas de recuperación de información, *precision* en inglés, no debe confundirse con la medida de precisión, *accuracy*, empleada para evaluar clasificadores. La precisión de un sistema de recuperación de información nos indica, de los documentos devueltos como resultado a una consulta, cuántos son realmente relevantes para esa consulta. En términos de nuestra matriz de contingencia:

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

Para complementar esta medida, que podríamos intentar maximizar reduciendo el número de documentos devueltos por el sistema, viene bien que recurramos a una medida complementaria: la exhaustividad o *recall*. Esta medida indica, de los documentos indexados que son realmente relevantes para nuestra consulta, cuántos se ofrecen en los resultados de la búsqueda; esto es, cómo de completos son los resultados de la búsqueda a la hora de encontrar los documentos relevantes de nuestra base de datos. Formalmente:

$$\text{recall} = \frac{TP}{TP + FN} = \frac{TP}{P}$$

En otros contextos, la medida de exhaustividad o *recall* también recibe el nombre de sensibilidad [*sensitivity*] o tasa de reconocimiento de verdaderos positivos [*TPR: True Positive Recognition rate*], algo que no resulta difícil de deducir si nos fijamos en su expresión aritmética:

$$TPR = \text{sensitivity} = \text{recall} = \frac{TP}{TP + FN} = \frac{TP}{P}$$

		Predicción	
		P	N
Clase real	P	TP	FN
	N	FP	TN

Figura 9: Precisión en un sistema de recuperación de información.

		Predicción	
		P	N
Clase real	P	TP	FN
	N	FP	TN

Figura 10: La medida de *recall* en un sistema de recuperación de información.

De la misma forma, podemos definir una tasa de reconocimiento de verdaderos negativos [*TNR: True Negative Recognition Rate*], que suele recibir el nombre de especificidad [*specificity*]:

$$TNR = \text{specificity} = \frac{TN}{TN + FP} = \frac{TN}{N}$$

Como podemos apreciar, mientras que la precisión (en sistemas de recuperación de información) se centra en las predicciones positivas (aciertos de la clase positiva, TP, y falsos positivos, FP), la exhaustividad o *recall* sesga nuestra percepción fijándose en la completitud de los resultados (aciertos de la clase positiva, TP, y falsos negativos, FN). La primera penaliza los falsos positivos. La segunda penaliza los falsos negativos.

En realidad, se trata de un problema de optimización multiobjetivo en el que la mejora de uno de nuestros objetivos se consigue siempre a costa del otro. Podemos reducir el número de falsos positivos (aumentar la precisión) reduciendo el número de resultados positivos, pero eso lo haremos a costa de reducir la exhaustividad de los resultados, ya que aumentaría el número de falsos negativos. Del mismo modo, podríamos reducir el número de falsos negativos (aumentar el *recall*) aumentando el número de resultados positivos, pero con ello sólo lograríamos reducir la precisión al colarse más falsos positivos en los resultados.

¿Y si nos interesa obtener una única medida que combine ambos criterios? Podríamos pensar que, con hacer la media aritmética entre las dos, sería suficiente. Sin embargo, eso podría inducirnos a pensar que una solución es aceptable por ser excelente en un sentido e ignorar por completo el otro. Si tenemos cierta probabilidad de acertar en la primera posición de la lista, devolviendo sólo ésta podríamos aspirar a una precisión del 100 % a costa de una exhaustividad pésima. O podríamos darle al usuario millones de resultados y decirle “ahí está todo”: *recall* 100 %, precisión nula.

Si de verdad queremos combinar precisión y *recall* en una única métrica, lo habitual es utilizar la media armónica de ambas. A diferencia de la media aritmética, la media armónica penalizará el hecho de que uno de nuestros objetivos tenga un valor bajo. Esa media armónica recibe el nombre de medida F o *F-score* (a saber por qué no la llamaron H). Se calcula de la siguiente forma a partir de la precisión y el *recall*:

$$F = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

De forma alternativa, haciendo algo de aritmética, también podemos expresar su valor en términos de las entradas de nuestra matriz de confusión:

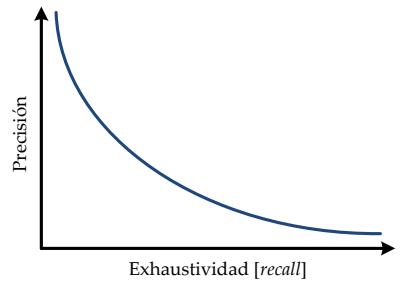


Figura 11: El compromiso entre precisión y *recall* (exhaustividad).

$$F = \frac{2TP}{2TP + FP + FN}$$

Además de tener en cuenta los aciertos positivos del sistema, como todas las medidas ya vistas, incorpora en una única métrica tanto los falsos positivos (precisión) como los falsos negativos (*recall*).

Imaginemos que tenemos dos alternativas para implementar un sistema, una de ellas ofrece un 0.6 tanto de precisión como de *recall*. La segunda ofrece un 0.9 de *recall* pero sólo un 0.3 de precisión. Desde el punto de vista de la media aritmética de las dos medidas individuales, el resultado es el mismo para ambos sistemas: un escaso 0.6. Sin embargo, mientras que el *F-score* del primer sistema se mantiene en el 0.6, el *F-score* del segundo baja a 0.45 ($2*0.3*0.9/(0.3+0.9)$), lo que nos haría decantarnos por la opción más equilibrada frente a la que optimiza uno de nuestros objetivos a costa de degradar excesivamente el otro.

El *F-score* que acabamos de presentar le da la misma importancia a la precisión y al *recall*. Pero se puede generalizar para ajustar la importancia relativa de ambos. La fórmula general, que incluye un parámetro β positivo, es la siguiente:

$$F_\beta = \frac{(1 + \beta^2) * precision * recall}{\beta^2 * precision + recall}$$

En ella, el parámetro β lo podemos utilizar para darle más importancia o menos importancia al *recall* frente a la precisión. La medida F_1 es el *F-score* tradicional, también conocido como coeficiente de similitud de Dice [*DSC: Dice Similarity Coefficient*]. Si representamos F_β en términos de las entradas de la matriz de contingencia, podemos ver mejor cuál es el efecto del parámetro β :

$$F_\beta = \frac{(1 + \beta^2)TP}{(1 + \beta^2)TP + FP + \beta^2FN}$$

F_β le da β veces más importancia al *recall* frente a la precisión. F_2 le da más importancia al *recall* penalizando más los falsos negativos, mientras que $F_{0.5}$ le da más importancia a la precisión atenuando la influencia de los falsos negativos. F_0 sería la precisión, F_∞ el *recall*.

Otra forma de integrar falsos positivos y falsos negativos en una única métrica es el coeficiente de correlación de Matthews, usado por el bioquímico del mismo nombre en 1975.⁵¹ También conocido como coeficiente ϕ (phi) en Estadística, o r_ϕ , resulta de aplicar el coeficiente de correlación de Pearson a la tabla de contingencia para medir la correlación entre dos variables binarias. Matemáticamente, el coeficiente de correlación de Matthews se define como:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Clase real	Predicción	
	P	N
P	TP	FN
N	FP	TN

Figura 12: *F-score*: Combinación de precisión y *recall* (exhaustividad) en una única medida.

⁵¹ B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442 – 451, 1975. ISSN 0005-2795. doi: 10.1016/0005-2795(75)90109-9

Como coeficiente de correlación, su valor estará siempre entre -1 y +1. Si cualquiera de los términos del denominador es cero, al denominador se le asigna arbitrariamente el valor 1, lo que resulta en $MCC = 0$.

Una métrica más que combina sensibilidad, TPR, y especificidad, TNR, es el índice de Youden.⁵² Conocido como estadístico J, se puede calcular fácilmente y también devuelve un valor entre -1 y +1:

$$J = TPR + TNR - 1$$

Propuesto originalmente para medir la efectividad de un test diagnóstico, devuelve un valor 0 cuando el test resulta en la misma proporción de resultados positivos para los grupos con y sin la enfermedad; esto es, cuando su valor diagnóstico es nulo. En otros contextos, recibe el nombre de estadístico *deltap'* y su generalización para problemas con más de dos clases se conoce con el sobrenombre de información del corredor de apuestas [*bookmaker informedness*].

¿Y si sólo nos interesa saber qué lugar ocupa en la posición la primera respuesta correcta de la lista? Esto nos podría interesar para evaluar ordenaciones alternativas en sistemas de recuperación de información y para medir la efectividad real de un sistema de QA utilizado en un asistente digital. En ese caso, realizaríamos una batería de pruebas. Para cada prueba, consideraríamos la lista ordenada de respuestas candidatas y evaluaríamos la lista en función de la posición que ocupe la primera respuesta correcta de las incluidas en la lista. Podríamos calcular, para nuestra batería de experimentos, la posición media de la primera respuesta correcta y utilizar ese indicador como medida de evaluación. Sin embargo, suele preferirse el uso de métricas acotadas, por lo que evaluaremos la bondad de una lista en la que la primera respuesta aparece en la posición *rank* usando el recíproco de esta posición: $1/rank$.

Con una simple media aritmética, podemos combinar los resultados de las N pruebas que realicemos y obtener como resultado una medida conocida por el acrónimo MRR [*Mean Reciprocal Rank*]:

$$MRR = \frac{1}{N} \sum_{i=1}^N \frac{1}{rank_i}$$

Un poco más arriba vimos que podíamos interpretar el *recall* como una tasa de verdaderos positivos, TPR:

$$TPR = sensitivity = recall = \frac{TP}{TP + FN} = \frac{TP}{P}$$

Del mismo modo, podemos definir una tasa de falsos positivos, FPR [*False Positive Rate*], también conocida como *fall-out*:

⁵² W. J. Youden. Index for rating diagnostic tests. *Cancer*, 3(1):32–35, 1950. ISSN 1097-0142. DOI: 10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3

El término *fall-out* hace referencia a los efectos secundarios adversos de una situación. Más concretamente, a las partículas radiactivas que se dispersan por la atmósfera tras un accidente o una explosión nuclear, las cuales terminan cayendo a la superficie en forma de partículas de polvo o como parte de las precipitaciones: la lluvia radiactiva.



$$FPR = \text{fall-out} = \frac{FP}{FP + TN} = \frac{FP}{N}$$

En términos de recuperación de información, la tasa FPR representa la probabilidad de que un documento no relevante sea incluido entre los documentos devueltos al realizar una consulta. En problemas de clasificación binarios, FPR no es más que la medida complementaria a la especificidad o TNR, como fácilmente puede comprobar:

$$FPR = \text{fall-out} = 1 - TNR = 1 - \text{sensitivity}$$

Si representamos de forma gráfica la tasa de verdaderos positivos TPR en el eje vertical y la tasa de falsos positivos FPR en el eje horizontal, obtenemos lo que se conoce con el nombre de curva ROC [*Receiver Operating Characteristic*]. Su curioso nombre se debe a que fueron diseñadas en los años 50 para analizar la recepción de señales con ruido (señales de radio). La misma técnica nos permite visualizar gráficamente el compromiso existente entre los aciertos (verdaderos positivos, TP) y las falsas alarmas (falsos positivos, FP) proporcionadas por un sistema.

Una curva ROC nos permite comparar visualmente el comportamiento relativo de distintos modelos de clasificación. Esta visualización puede resultar atractiva a la hora de presentar resultados en un entorno de negocios y hay quien hace especial hincapié en el uso de este tipo de representaciones visuales para mostrar los objetivos alcanzados en la construcción de un modelo de aprendizaje supervisado.⁵³ En cierto modo, la forma de la curva ROC ofrece una perspectiva más completa que una simple medida numérica.

La curva ROC siempre conecta los puntos (0,0) y (1,1). Dada una lista de ejemplos previamente etiquetados, utilizamos algún modelo de clasificación que nos permita predecir la probabilidad de que cada ejemplo E pertenezca a la clase positiva $\hat{p}(+|E)$. A continuación, ordenamos los ejemplos en orden decreciente del valor estimado $\hat{p}(+|E)$. Al recorrer la lista de ejemplos ordenados, si nos encontramos un ejemplo que pertenece

Figura 13: Curvas ROC, tal como aparecían en las camisetas que Amazon regalaba a los asistentes al congreso de minería de datos organizado anualmente por la ACM (en concreto, en el KDD de Chicago). Seguramente, uno de los pocos lugares del mundo donde se podría entender algo así. En el otro lado de la camiseta aparecía la siguiente frase: “Tenemos los datos... ¿puedes leer mentes?”.

⁵³ Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, 2012. ISBN 1107422221

a la clase positiva (un posible acierto), nos movemos verticalmente hacia arriba. Si nos encontramos un ejemplo de la clase negativa (un posible error), nos movemos hacia la derecha.

El clasificador perfecto colocaría todos los ejemplos de la clase positiva antes de los ejemplos de la clase negativa, por lo que la curva ROC iría del origen $(0,0)$ directamente hasta el $(0,1)$, y de ahí hacia la derecha hasta el punto $(1,1)$. Si nuestro clasificador no es demasiado bueno y, en la lista ordenada por probabilidad decreciente, tenemos la misma probabilidad de encontrarnos un ejemplo de la clase positiva y un ejemplo de la clase negativa, la curva ROC asociada a nuestro clasificador seguirá la diagonal desde el origen $(0,0)$ hasta el punto $(1,1)$.

¿Qué representa realmente la curva ROC? En realidad, la curva ROC representa gráficamente el comportamiento de nuestro clasificador cuando utilicemos diferentes umbrales de la probabilidad de $\hat{p}(+|E)$, que representa la probabilidad estimada de que un ejemplo E pertenezca a la clase positiva de nuestro problema de clasificación. Si quisieramos construir un clasificador binario, utilizaríamos un umbral U para $\hat{p}(+|E)$: los ejemplos con $\hat{p}(+|E) \geq U$ diríamos que son de la clase positiva y los ejemplos con $\hat{p}(+|E) < U$ diríamos que son de la clase negativa. Distintos valores para ese umbral darían lugar a clasificadores diferentes. Cada clasificador tendría una tasa de aciertos TPR y una tasa de falsos positivos FPR diferente, en función del punto sobre la curva ROC en el que nos encontrásemos (la precisión, obviamente, iría disminuyendo conforme aumentásemos el *recall*). La curva ROC resume visualmente todas las opciones que tenemos a la hora de establecer ese umbral que determine cuándo consideramos que un ejemplo es de una clase o de la otra.

El área que queda debajo de la curva ROC es una medida de la precisión del clasificador. Dicha área, que recibe el original nombre de área bajo la curva o AUC [*Area Under Curve*], es otra medida que se utiliza a menudo para presentar los resultados obtenidos con un modelo de aprendizaje supervisado. Un modelo de clasificación perfecto, obviamente, tendrá área uno: $AUC = 1$. Un clasificador como el del párrafo anterior, en el que la curva ROC era una diagonal, tendrá área $AUC = 1/2$. Cuanto más cerca esté la curva ROC de nuestro clasificador de la diagonal, menos preciso será nuestro modelo de clasificación.

Un buen valor de AUC no indica necesariamente que nuestro modelo sea bueno. Igual que sucedía con la precisión con la que empezamos esta sección de métricas de evaluación, una interpretación correcta de los resultados dependerá de las implicaciones que tengan, en nuestro problema particular, las apariciones de falsos positivos y de los falsos negativos. Es la forma de la curva ROC lo que realmente importa.

Por ejemplo, supongamos que tenemos un modelo con $AUC = 0.76$, relativamente bueno. Sin embargo, ese valor puede deberse a que nuestro

modelo de clasificación es especialmente bueno para el primer cuartil de los datos, en los que la curva ROC es prácticamente vertical. A partir de ese momento, nuestra curva es, a efectos prácticos, paralela a la diagonal asociada a un clasificador que no discrimina bien entre las dos clases. Si nos fiamos demasiado del valor de AUC, podemos pensar que nuestro clasificador es especialmente bueno cuando, en realidad, sólo funciona bien para el primer cuarto de los datos (y, supongamos, que también para el último). Si los errores de clasificación tienen un coste económico asociado, es importante que lo tengamos en cuenta. Si en un entorno empresarial, estamos invirtiendo recursos en captar clientes, el primer cuartil seleccionado por nuestro clasificador nos ofrece unos retornos espectaculares (todo lo que invertimos revierte en futuros ingresos para nuestra empresa). Sin embargo, a partir de ahí, la inversión que realizamos es prácticamente equivalente a jugar a los dados. El modelo no nos aporta nada en la zona de su curva ROC paralela a la diagonal.

La moraleja de todo esto es que un modelo aparentemente bueno puede que no lo sea tanto (el ejemplo del párrafo anterior). De la misma forma, un modelo aparentemente malo puede que sea especialmente bueno a la hora de identificar los ejemplos más prometedores, aquellos con mayor $\hat{p}(+|E)$, aunque su comportamiento sea peor que lamentable para el resto.

En términos de marketing, no todos los ‘leads’ son iguales: para los mejores clientes potenciales (leads), el coste de conversión puede merecer la pena; para otros, el coste de conversión puede ser demasiado elevado.

Si sólo queremos identificar a los más prometedores, algo habitual en muchas aplicaciones, nos puede bastar un ‘mal’ modelo, con AUC bajo, pero un inicio de la curva ROC casi vertical. Si de verdad tenemos que establecer un ranking global, entonces puede que nos interese recurrir a alguna técnica cuyo AUC no sea el mejor, pero que ofrezca un comportamiento más robusto en la zona media de su curva ROC.

Predicciones cuantitativas

No todos los problemas de aprendizaje son problemas de clasificación, en los que se intenta predecir el valor de una variable categórica. En ocasiones, hemos de construir modelos capaces de predecir el valor de una variable numérica, que toma valores dentro de un rango continuo. Se trata de un problema habitual en Estadística para el que se han propuesto multitud de técnicas de regresión. Otro ámbito de aplicación en el que resulta necesaria la predicción de una variable continua es la predicción de series temporales.

En la predicción de valores de tipo numérico, nuestro modelo de aprendizaje será de la forma $\hat{y} = f(x)$. A partir de un conjunto de datos x , intentaremos estimar el valor de y por medio de una función $f(x)$. En

un modelo de regresión, los valores asociados a x incluirán todas aquellas variables que hayamos decidido considerar en nuestro modelo. En un modelo de predicción de series temporales, los valores x corresponden a los valores que la serie temporal tomó en el pasado y cuyo futuro queremos predecir utilizando $f(x)$.

La medida de error más utilizada para estimar la calidad de un modelo de predicción cuantitativa es el error cuadrático medio [*MSE: Mean Squared Error*], que aplicamos sobre las N muestras del conjunto de datos sobre el que estamos estimando la calidad de nuestro modelo cuantitativo:

$$MSE = \frac{1}{N} \sum_{i=1}^N (f(x) - y)^2$$

En ocasiones, ni siquiera nos molestaremos en calcular la media, sino que nos valdrá la suma de los errores al cuadrado [*SSE: Sum of Squared Errors*]:

$$SSE = N * MSE = \sum_{i=1}^N (f(x) - y)^2$$

A la hora de comparar alternativas, si el valor de N no cambia, nos da igual emplear MSE o SSE. Por ejemplo, muchos algoritmos de clustering requieren establecer de antemano el número de clusters k que deseamos encontrar en nuestro conjunto de datos, como en el algoritmo de las k medias o *k-means*. Dado que el valor adecuado de k no lo solemos conocer de antemano, podemos probar con distintos valores de k y ver cuál es el resultado que se obtiene. En el caso de algoritmos como *k-means*, podemos calcular una medida de error SSE sumando las distancias de cada punto de nuestro conjunto de datos al centroide del cluster al que ha sido asignado. Obviamente, si tuviésemos tantos clusters como datos en nuestro conjunto, la medida de error SSE sería cero, más o menos la misma utilidad que tendrían los resultados del algoritmo de agrupamiento. Sin embargo, podemos representar gráficamente los valores de SSE (o MSE, si se prefiere) y obtendremos una función decreciente para valores crecientes de k . Idealmente, la función $SSE(k)$ tendrá forma de L y la posición del codo de esa L nos sugerirá el número de clusters adecuado para nuestro conjunto de datos. En teoría, ya que no siempre se observa claramente un codo en esa función.

En el análisis estadístico de datos, se suele emplear la desviación típica σ en lugar de la varianza σ^2 porque la desviación se mide en las mismas unidades que la variable que estamos analizando, lo que facilita la interpretación física de los resultados de nuestro análisis. Análogamente, podemos calcular la raíz cuadrada del error cuadrático medio para obtener

Una serie temporal no es más que una secuencia de datos, observaciones o valores, medidos en determinados instantes de tiempo y ordenados cronológicamente. A menudo, los datos de una serie temporal están espaciados de forma regular en el tiempo; esto es, se registran periódicamente valores para años, trimestres, meses, semanas, días, horas, minutos, segundos o femtosegundos consecutivos, según la escala temporal a la que deseemos analizar un fenómeno.

En realidad, el algoritmo *k-means* minimiza el error SSE si utilizamos la distancia euclídea como medida de distancia y la media aritmética para calcular los centroides. Si optamos por utilizar la distancia de Manhattan, el algoritmo *k-means* minimiza el error SSE siempre que utilicemos la mediana para calcular la posición de los centroides.

una medida de error denominada RMSE [*Root Mean Squared Error*]:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N (f(x) - y)^2}$$

RMSE, a veces representada por RMSD [*Root Mean Squared Deviation*] representa la desviación de las diferencias entre los valores de las predicciones $f(x)$ y los valores observados y . Esas diferencias se denominan residuos cuando se están calculando sobre el conjunto de datos empleado para construir el modelo. Cuando se calculan sobre otros conjuntos de datos, se llaman errores de predicción.

SSE, MSE y RMSE sirven para agregar en una única medida las magnitudes de los errores cometidos sobre un conjunto de datos, por lo que desempeñan un valor similar a la precisión o la tasa de error en los problemas de clasificación. Nos permiten comparar los errores cometidos por modelos alternativos para un mismo conjunto de predicciones, pero no para comparar predicciones sobre distintos conjuntos de datos, al ser dependientes de la escala de los datos.

Estas medidas de error se pueden normalizar para comparar modelos sobre conjuntos de datos que se representen sobre escalas diferentes. Aunque no hay una normalización particular que se considere estándar, podemos normalizar una medida como RMSE utilizando, o bien la media aritmética, o bien el rango de los valores de la variable sobre la que estamos midiendo el error:

$$NRMSE_{range} = \frac{RMSE}{y_{max} - y_{min}}$$

o bien

$$NRMSE_{mean} = \frac{RMSE}{\bar{y}}$$

Todas las medidas vistas hasta ahora incorporan en sus fórmulas el cuadrado del error para los errores cometidos. ¿Por qué el cuadrado y no simplemente la diferencia entre la estimación y el valor observado? Porque usando directamente las diferencias $f(x) - y$, los errores por encima y por debajo se compensarían, con lo que las medidas de error no servirían para nada.

El problema de utilizar los errores al cuadrado es que hace que nuestras medidas sean sensibles a la presencia de anomalías u *outliers* en los datos. El efecto de cada error individual es proporcional al cuadrado del tamaño del error, por lo que los errores más grandes (p.ej. los asociados a los outliers) tienen efectos desproporcionados sobre medidas de error como SSE, MSE o RMSE.

Por este motivo, se suele recomendar el uso del error absoluto medio en lugar de medidas como RMSE. El error absoluto medio o MAE [*Mean*

Absolute Error] se define de la siguiente forma:

$$MAE = \frac{1}{N} \sum_{i=1}^N |f(x) - y|$$

El error MAE es más sencillo de interpretar que el error RMSE, al no ser más que la media de las diferencias, en valor absoluto, entre las predicciones $f(x)$ y las observaciones y . Además, a diferencia de RMSE, la contribución directa de cada error individual es proporcional al valor absoluto del error, lo que hace al error MAE más robusto frente a la presencia de ruido y *outliers* en el conjunto de datos.

Igual que sucedía con el error RMSE, existen distintas formas de normalizar el error absoluto medio para poder establecer comparaciones entre los resultados obtenidos sobre diferentes conjuntos de datos. Podemos añadir algunas de ellas a nuestra colección de acrónimos: MAPE [*Mean Absolute Percentage Error*], sMAPE [*Symmetric MAPE*], MASE [*Mean Absolute Scaled Error*], MRAE [*Mean Relative Absolute Error*], MdRAE [*Median Relative Absolute Error*], UMBRAE [*Unscaled Mean Bounded Relative Absolute Error*]... No terminaríamos nunca.

El rasgo común de todas ellas es que dividimos los errores observados, las diferencias $f(x) - y$ en valor absoluto, por una cantidad que podemos definir en función de nuestros objetivos concretos. Sólo debemos asegurarnos de evitar la posibilidad de que, para determinados conjuntos de datos, la normalización acabe realizando una división por cero, lo que haría inservible nuestra métrica de error.

La medida normalizada de error absoluto más conocida tal vez sea el error absoluto medio porcentual MAPE [*Mean Absolute Percentage Error*], también conocido como desviación absoluta media porcentual MAPD [*Mean Absolute Percentage Deviation*]. Se define de la siguiente forma:

$$MAPE = \frac{100}{N} \sum_{i=1}^N \frac{|f(x) - y|}{|y|}$$

Básicamente, MAPE nos da el error medio cometido en la estimación $f(x)$ expresado como un tanto por ciento sobre el valor observado de y . Aunque su definición parece convincente, tiene algunos inconvenientes. En primer lugar, no puede utilizarse si, en ocasiones, la variable analizada y toma el valor cero. Además, la medida MAPE es asimétrica en el siguiente sentido: el error de una predicción que subestime el valor real y no puede superar el 100 % pero el porcentaje de error no está acotado si sobreestimamos el valor de y . Es decir, si la utilizamos para comparar entre modelos predictivos alternativos, MAPE está sesgada a favor de las técnicas de predicción que subestimen sistemáticamente el valor que intentamos predecir.

Este problema se podría solucionar si utilizamos una medida relativa que no tenga ese sesgo a favor de las subestimaciones, p.ej. la razón $\log(f(x)/y)$, que es simétrica para sobreestimaciones y subestimaciones.

Los problemas causados por la asimetría de MAPE se han intentado resolver proponiendo varias versiones simétricas de MAPE. Aunque existen definiciones inconsistentes de esta medida en distintas publicaciones, algunas de ellas con errores, podríamos definir sMAPE [*symmetric MAPE*] como

$$sMAPE = \frac{100}{N} \sum_{i=1}^N \frac{2|f(x) - y|}{|f(x)| + |y|}$$

Algunas versiones de sMAPE no están acotadas. La mostrada aquí tiene un rango del 0 al 200 % siempre y cuando $f(x)$ e y no sean simultáneamente 0.

Para apreciar mejor las diferencias entre MAPE y sMAPE, utilicemos un ejemplo. Supongamos que, para un dato dado, su valor observado es $y = 100$. Un modelo de predicción f_1 realiza una predicción $f_1(x) = 110$, mientras que un segundo modelo f_2 predice $f_2(x) = 90$. Sobre la escala de porcentajes a la que estamos habituados, MAPE es simétrico y, en ambos casos nos da un error MAPE del 10 %. Sin embargo, la versión “simétrica” de MAPE, sMAPE, es asimétrica al medir los porcentajes de error: $sMAPE(f_1) = 9.52\%$ y $sMAPE(f_2) = 10.53\%$. La asimetría que elimina sMAPE es la asociada al sesgo de MAPE a favor de las subestimaciones.

El experto australiano en predicción de series temporales Rob J. Hyndman, de la Monash University en Melbourne, recomienda utilizar la medida MAPE original (más sencilla de interpretar cuando tiene sentido usarla) o recurrir a una medida alternativa denominada MASE [*Mean Absolute Scaled Error*], que él mismo propuso en 2005:

$$MASE = \frac{1}{N} \sum_{i=1}^N \frac{|f(x) - y|}{Q}$$

con Q fijado en función de la escala de la serie temporal (y de si es estacional o no). Esta medida se diseñó para que reuniese una serie de características deseables que no reúnen las medidas anteriores: invarianza frente a cambios de escala (permite comparar predicciones sobre distintos conjuntos de datos), comportamiento predecible para valores de y cercanos a cero (evita la división por cero de MAPE), simetría (penaliza los errores positivos y negativos de la misma forma, a diferencia de sMAPE) e interpretabilidad (el valor Q se escoge para que valores de MASE superiores a 1 correspondan a predicciones peores que las que se podrían realizar de forma directa a partir de las muestras anteriores).

Otras medidas

Aunque usualmente estaremos interesados en conseguir modelos de aprendizaje lo más precisos posible, la precisión de un modelo no siempre será el único criterio que utilizaremos para evaluar la calidad de los modelos construidos. Cobertura, robustez, estabilidad, interpretabilidad, complejidad, eficiencia, escalabilidad o, incluso, serendipia pueden ser relevantes en el problema particular que pretendamos resolver.

- *Cobertura*

La cobertura [*coverage*] de un modelo hace referencia a su capacidad de aplicación sobre los datos que se puedan presentar. En ocasiones, un modelo puede que sea incapaz de tomar una decisión que ofrezca un mínimo de garantías, por lo que se negará a proporcionarnos una respuesta que podamos utilizar en esos casos. Si lo que pretendemos hacer es automatizar un proceso que se venía realizando de forma manual, la cobertura nos indica el grado de automatización que hemos alcanzado con la ayuda de las técnicas de aprendizaje automático. Si lo que estamos haciendo es diseñar un sistema de recomendación, la cobertura nos indica cuántos ítems es capaz el sistema de incluir en sus recomendaciones (y, por consiguiente, cuántos no aparecerán nunca en las recomendaciones que es capaz de realizar).

- *Robustez*

La robustez de un modelo también es un factor a tener en cuenta. Dado que, en la práctica, los datos con los que ha de trabajar un modelo siempre contienen errores, vienen acompañados de ruido y pueden incluir valores nulos, es importante que la técnica de aprendizaje que utilicemos para construir un modelo sea capaz de funcionar correctamente en situaciones como las que se encontrará en el mundo real.

- *Estabilidad*

La estabilidad de nuestros modelos puede resultar fundamental en sistemas que interactúan con seres humanos (o con otros sistemas diseñados de forma maliciosa por otros seres humanos). Por ejemplo, las recomendaciones ofrecidas por un sistema de recomendación o los resultados de un sistema de recuperación de información no deberían verse demasiado afectados porque un atacante pretenda modificarlos. En ocasiones, existen incentivos económicos que pueden empujar a determinadas personas a intentar manipular el funcionamiento normal de un sistema construido con la ayuda de técnicas de aprendizaje automático, por lo que es otra faceta del sistema que deberemos tener en cuenta.

■ *Interpretabilidad*

La interpretabilidad de un modelo puede ser otro aspecto relevante en la práctica, especialmente si nuestro puesto de trabajo depende de la misma después de que el sistema haya cometido un error, aparentemente, garrafal. En ciertas aplicaciones, nos dará igual si nuestro sistema se comporta como una caja negra, siempre y cuando ofrezca resultados aceptables en cuanto a su precisión, tasa de cobertura, robustez o escalabilidad. Sin embargo, en determinados ámbitos, la interpretabilidad del modelo puede ser un requisito *sine qua non* para que podamos utilizarlo en la práctica, puede que incluso por obligaciones legales. La Unión Europea, por ejemplo, pretende implantar una regulación de protección de datos según la cual los usuarios tendrían el derecho a recibir una explicación de las decisiones tomadas de forma algorítmica.

Dado que las redes neuronales artificiales no destacan precisamente por su interpretabilidad, ¿será este el fin del deep learning en Europa?

■ *Complejidad*

Relacionada en ocasiones con su interpretabilidad, la complejidad de un modelo es otra característica que puede ser importante para nosotros. Usualmente, los métodos de aprendizaje suelen incluir algún tipo de sesgo que les lleva a preferir soluciones sencillas frente a alternativas más complejas que ofrezcan resultados similares. Es la llamada navaja de Occam, el principio filosófico de parsimonia según el cual, en igualdad de condiciones, se asume que la explicación más sencilla es también la más probable.

Desde otro punto de vista, como veremos cuando hablamos del compromiso entre sesgo y varianza común a muchas técnicas de aprendizaje, la complejidad del modelo puede influir decisivamente en su calidad. Un modelo demasiado simple para el sistema que pretendemos modelar será poco versátil en la práctica porque incorporará, de serie, demasiadas restricciones. En el extremo opuesto, un modelo más complejo de lo necesario para nuestro sistema, con más parámetros que ajustar que datos disponibles en el conjunto de entrenamiento, tenderá a reflejar demasiados matices presentes en el conjunto de entrenamiento que luego no resultan útiles a la hora de generalizar.

■ *Eficiencia*

Ya desde un punto de vista eminentemente práctico, la eficiencia de los algoritmos necesarios para construir nuestro modelo es vital para que una solución pueda ser factible. El tipo de técnica de aprendizaje suele determinar el tiempo necesario, en primer lugar, para construir el modelo a partir de un conjunto de datos y, a continuación, para utilizar el modelo en la práctica. Si la construcción del modelo es algo que sólo necesitamos hacer una vez, tal vez podamos permitirnos el lujo de emplear una técnica de aprendizaje que resulte especialmente

Desde el punto de vista del autor, algo más absurdo aún que el llamado derecho al olvido, del que sólo se suelen beneficiar políticos corruptos y otros tipos de delincuentes, a los cuales les ofrece un mecanismo para que un usuario normal no pueda acceder a información ya publicada en medios de comunicación a través de un buscador web.

costosa. Si el sistema debe tomar decisiones en tiempo real una vez puesto en marcha, tal vez tengamos que olvidarnos de ciertos métodos de aprendizaje cuyos requisitos técnicos impiden su funcionamiento en tiempo real utilizando el hardware que tengamos disponible.

- *Escalabilidad*

Relacionada con la eficiencia, la escalabilidad de una técnica concreta de aprendizaje determina si podemos aplicarla sobre los conjuntos de datos enormes con los que hoy se trabaja en *big data*. Aunque el coste computacional asociado a un algoritmo de aprendizaje sea lineal con respecto al tamaño del conjunto de datos de entrenamiento, ese algoritmo puede que no sea viable en la práctica si no podemos paralelizarlo de alguna forma. Si disponemos de hardware especializado, como es el caso de las GPUs [*Graphical Processing Units*] que paralelizan la ejecución de operaciones de cálculo matricial, tal vez podamos aprovechar las ventajas que ofrece el deep learning.

Los anteriores son sólo algunos de los aspectos que podemos tener en cuenta a la hora de evaluar modelos de aprendizaje, tanto supervisado como no supervisado. Según el contexto en el que nos encontremos, tal vez aparezcan factores adicionales que debamos considerar a la hora de decidirnos por un modelo u otro.

- En un sistema de recomendación pueden resultar de interés facetas como la diversidad (que las recomendaciones incluyan elementos seleccionados de acuerdo a diferentes criterios, para que no resulten excesivamente uniformes), la novedad (capacidad del sistema de ofrecerle a un usuario recomendaciones de artículos o productos que no haya visto antes) o la serendipia (que el sistema sea capaz de sorprender con sus recomendaciones, sugiriendo conexiones inesperadas que puedan despertar el interés del usuario).
- En un algoritmo de clustering, puede que nos interese obtener resultados que cumplan determinadas propiedades con respecto a la cohesión, separación o silueta de los clusters. El grado de cohesión nos ayudará a reducir la distancia intra-cluster. El grado de separación aumentará conforme aumente la distancia inter-cluster. Un coeficiente de silueta [*silhouette coefficient*] puede servirnos para validar la consistencia de los clusters identificados: cómo de similar es un objeto con los elementos de su cluster (cohesión) con respecto a los miembros de otros clusters (separación). En función de lo que deseemos obtener, se pueden definir múltiples medidas que intenten representar características deseables de un conjunto de clusters, la interconectividad interna de un cluster, la interconectividad entre un par de clusters o la interconectividad relativa entre ambos. Muchas de esas medidas suelen aparecer propuestas junto a algoritmos concretos de agrupamiento

y, casualmente, suelen ser esos algoritmos los que mejores resultados consiguen con respecto a las medidas propuestas con ellos.

Como hemos visto, existe infinidad de criterios que podemos utilizar a la hora de evaluar la calidad de los resultados proporcionados por un modelo de aprendizaje automático. Cada una de las muchas técnicas de aprendizaje automático que se han propuesto intenta optimizar su rendimiento con respecto a alguno de esos criterios. A la hora de elegir una métrica en concreto, no obstante, deberemos evaluar cuáles son nuestras prioridades concretas.

Puede que tengamos restricciones en cuanto al uso que podemos hacer de distintos tipos de recursos, como tiempo, espacio, uso de energía o dinero. Nuestro tiempo puede estar limitado por nuestra capacidad de atención al problema concreto mientras atendemos otras obligaciones. El tiempo de CPU disponible para ejecutar algoritmos puede estar limitado por los recursos computacionales de los que dispongamos y las fechas de entrega a las que nos hayamos comprometido. El uso de energía del sistema puede ser relevante si estamos diseñando un sistema que deba funcionar en dispositivos móviles, cuya alimentación depende del uso de baterías. Nuestras limitaciones económicas pueden determinar la escala de nuestros experimentos, especialmente si aprovechamos la infraestructura proporcionada por un proveedor de *cloud computing* para la ejecución de nuestros algoritmos.

Por otro lado, puede que nos enfrentemos a problemas en los que exista cierto grado de incertidumbre, desde información incompleta y datos erróneos hasta factores desconocidos de los que ni siquiera seamos conscientes (los temidos “*unknown unknowns*”).

En definitiva, no existen criterios objetivos que nos permitan indicar de antemano cuáles deben ser las métricas que deberíamos utilizar en cada momento para evaluar nuestros modelos. Como decía Bertrand Russell, todo el conocimiento humano es incierto, inexacto y parcial. En aprendizaje automático, que trabaja siempre con heurísticas, aún más.

Métodos de evaluación

A la hora de evaluar el comportamiento de nuestro sistema, independientemente de las métricas concretas que hayamos seleccionado para hacerlo, debemos seguir ciertas directrices. Al construir un modelo, estamos intentando comprender algo que nunca conocemos del todo. No nos queda más remedio que estimar cómo de bueno es el modelo que estamos construyendo. Para realizar esa estimación, recurriremos a técnicas de tipo estadístico.

En aprendizaje automático, el modelo lo construimos a partir de un conjunto de datos. Ese conjunto de datos es todo lo que tenemos para

estimar la bondad del modelo y, dado que no disponemos de otros datos, ese conjunto de datos tendremos que utilizarlo para, además de construir el modelo, estimar su calidad.

El objetivo es que esa estimación nos sirva para evaluar, en media, cuál será el comportamiento del modelo una vez que nos pongamos a utilizarlo. Para ello podemos reservar una parte de los datos disponibles. En vez de utilizar todos los datos disponibles para entrenar el modelo, una parte no la utilizaremos en el entrenamiento. Será la que nos sirva para estimar cómo se comportará el modelo con datos diferentes a los datos con los que se ha construido. Es el llamado conjunto de prueba.

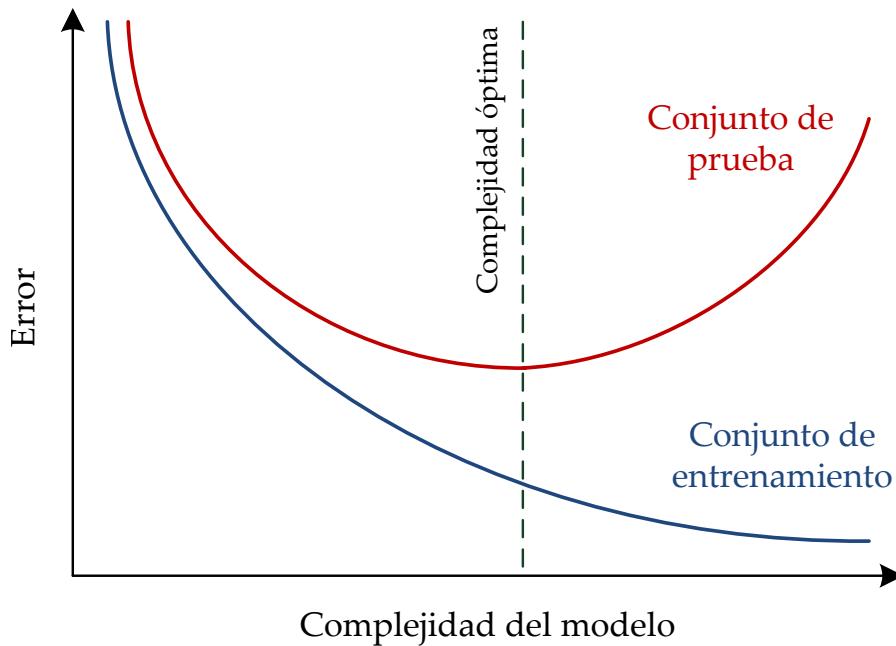
Por otro lado, también nos gustaría ser capaces de medir la varianza o la desviación asociada a nuestra estimación puntual de la bondad del modelo construido. Utilizar un conjunto de prueba separado del conjunto de entrenamiento nos puede servir para realizar una estimación puntual de forma rápida y sencilla. Pero desconocemos su varianza, que puede ser elevada, por lo que nunca estaremos del todo seguros acerca de cómo de bien se acabará comportando nuestro modelo en la práctica.

Algunas técnicas estadísticas nos pueden ayudar. Si realizamos varios experimentos, esos experimentos repetidos nos permitirán hacernos una idea de cuánto pueden variar las estimaciones puntuales obtenidas en cada experimento. La media de esas estimaciones nos puede servir para determinar cómo se comportará el modelo construido con conjuntos de datos distintos a los de su entrenamiento. La varianza o la desviación de las estimaciones puntuales derivadas de cada experimento nos servirá para descubrir si el rendimiento del sistema será siempre similar (varianza reducida) o si esperamos que se produzcan fluctuaciones (varianza elevada).

Las técnicas estadísticas de remuestreo [*resampling*] serán las que nos permitan realizar una estimación de algo inicialmente desconocido para nosotros: la precisión de muestras medidas. Esas técnicas seleccionan subconjuntos de datos [*jackknifing*] del conjunto de datos disponible para realizar una batería de experimentos que nos proporcionará estimaciones de medias y desviaciones para las métricas que estemos empleando para evaluar la calidad de nuestros modelos.

Cuando realicemos un muestreo sin reemplazo de los datos, hablaremos de validación cruzada. Cuando el muestreo aleatorio se realice con reemplazo, hablaremos de *bootstrapping*. Tanto la validación cruzada como el *bootstrapping* nos permitirán validar la bondad de los modelos obtenidos a partir de datos usando técnicas de aprendizaje automático.

Además, cuando dispongamos de técnicas alternativas para resolver un mismo problema, las herramientas de remuestreo nos servirán para guiarnos en nuestra selección de la técnica que resulte más adecuada en un contexto particular. Lo más habitual es que seleccionemos aquélla de mayor rendimiento, aunque también podríamos escoger aquélla que



muestra una menor varianza, lo que permitirá que el comportamiento del sistema sea un poco más predecible y algo menos errático, aun a costa de perder cierto grado de eficacia.

Conjuntos de entrenamiento, validación y prueba

Nunca se debe utilizar el conjunto de entrenamiento para evaluar la calidad de un modelo obtenido por técnicas de aprendizaje automático. Bajo ninguna circunstancia. Never!

Si, en un problema de aprendizaje supervisado, construimos un clasificador a partir de un conjunto de datos de entrenamiento, ese clasificador debería ser muy bueno para clasificar los datos del conjunto de entrenamiento. Suponiendo que la técnica de aprendizaje utilizada realmente aprenda algo a partir del conjunto de datos de entrenamiento, algo habitual en las técnicas que se suelen utilizar, al clasificar el conjunto de datos de entrenamiento obtendríamos una tasa de error: el “error de resustitución” del clasificador.

Una vez construido un modelo a partir del conjunto de entrenamiento, se usa dicho modelo sobre los datos del conjunto de prueba, que hasta ese momento nunca habrá visto. Comparando las etiquetas asociadas a los casos del conjunto de prueba con el resultado de aplicar el modelo, se obtiene fácilmente una medida de su precisión. Si la precisión del clasificador es aceptable sobre el conjunto de prueba, y sólo entonces, podremos utilizar el modelo en la práctica para clasificar nuevos casos de los que realmente desconocemos su clase.

Figura 14: El problema del sobreaprendizaje: error en el conjunto de entrenamiento vs. error en el conjunto de prueba.

Un problema muy habitual en aprendizaje automático es el problema del sobreaprendizaje [*overfitting*]. En general, para la mayor parte de las técnicas de aprendizaje automático, cuanto mayor sea la complejidad del modelo aprendido, más tenderá a ajustarse al conjunto de datos de entrenamiento utilizado en su construcción (de ahí lo de *overfitting*). Sin embargo, sobre datos reales, ese sobreajuste puede hacer que el modelo sea menos útil para trabajar con datos nuevos, como los que se encontrará cuando empecemos a utilizarlo. En otras palabras, generalizará peor. Ésa es la causa de que el error de clasificación en el conjunto de entrenamiento (i.e. el error de resustitución) no sea un buen estimador de la precisión del clasificador. Esa medida de error estará claramente sesgada: será excesivamente optimista. Por tanto, el error de resustitución no nos sirve, bajo ningún pretexto, para predecir el rendimiento de nuestro clasificador cuando se tenga que enfrentar a datos nuevos, necesariamente distintos a los datos que aparecen en su conjunto de entrenamiento.

El problema del sobreaprendizaje es el motivo por el que no podemos utilizar, para evaluar un modelo, el mismo conjunto de datos que utilizamos para construirlo. Bajo ningún concepto. Para evaluar la calidad de un modelo, debemos recurrir a un conjunto de prueba independiente del conjunto de entrenamiento utilizado en la construcción del modelo [*holdout*]. Por ejemplo, podemos reservar dos tercios de los datos disponibles para entrenar el modelo y el tercio restante lo utilizaríamos para estimar su calidad. La descomposición de 2/3 para el conjunto de entrenamiento y 1/3 para el conjunto de prueba no está escrita en piedra, por lo que tampoco pasaría nada si optamos por una descomposición 70 %-30 %. Lo importante es que no nos olvidemos de dividir el conjunto de datos disponible en un conjunto de entrenamiento (para construir el modelo) y un conjunto de prueba (para evaluar el modelo).

En un contexto ideal, cuantos más datos tengamos a nuestra disposición, mejor podremos entrenar nuestro modelo. Si utilizamos técnicas escalables, podremos aprovechar la disponibilidad de un conjunto de entrenamiento enorme para entrenar el modelo. Cuanto mayor sea el tamaño del conjunto de test, más precisas serán nuestras estimaciones de la calidad del modelo.

Obviamente, para que nuestra estimación de la calidad de un modelo sea realista, los conjuntos de datos de entrenamiento y prueba deben ser, además de independientes, ejemplos representativos de los datos asociados al problema que pretendemos resolver. Si diseñamos un clasificador que identifique señales de tráfico para un vehículo autónomo, no deberíamos esperar que ese clasificador vaya a funcionar bien como reconocedor facial en un sistema de seguridad biométrica.

En problemas de clasificación no balanceada, por ejemplo, unas clases son mucho más frecuentes que otras. Al descomponer el conjunto de datos en conjunto de entrenamiento y conjunto de prueba deberíamos

Al menos, no siempre. Existe una rama del aprendizaje automático, denominada aprendizaje por transferencia [*transfer learning*], que persigue precisamente ese objetivo. Reutilizar lo aprendido en un dominio para mejorar su rendimiento en otro diferente.

tenerlo en cuenta, para evitar que una clase poco frecuente pueda no estar representada en el conjunto de prueba. Al dividir nuestros datos, podemos estratificarlos: asegurarnos de que todas las clases están representadas proporcionalmente tanto en el conjunto de entrenamiento como en el conjunto de prueba.

La tasa de error en el conjunto de prueba es sólo una estimación de la tasa de error real del clasificador. Esa tasa real es desconocida para nosotros, ya que nos hemos limitados a utilizar una (pequeña) muestra de datos para estimar su valor: el conjunto de prueba. Además, esa muestra la hemos escogido al azar al descomponer el conjunto de datos disponible en conjunto de entrenamiento y conjunto de prueba. Sin querer, puede que hayamos introducido algún sesgo en el conjunto de prueba, lo que hará menos fiable nuestra estimación de la calidad del modelo.

Por si esto fuera poco, al no utilizar todos los datos disponibles para construir el modelo, puede que nuestro clasificador no sea todo lo bueno que podría llegar a ser (si hubiésemos empleado el conjunto de datos completo en su construcción). Llegamos a un callejón aparentemente sin salida: tenemos que reservar datos para evaluar el modelo pero el modelo sería mejor si no reservamos nada y usamos todos los datos a nuestra disposición. En la práctica, esta falsa contradicción se resuelve de una forma muy simple. Primero se realiza un experimento para estimar la calidad del modelo que podemos obtener. Una vez finalizada esa estimación, se desechan los modelos usados durante la fase de estimación y se construye un modelo final, que construimos a partir del conjunto de datos completo.

Un error muy común es reutilizar el mismo conjunto de prueba para múltiples experimentos en los que se van variando los parámetros que admiten las distintas técnicas de aprendizaje. El conjunto de prueba, pues, deja de ser independiente. Se podría decir que estamos entrenando modelos particulares con la ayuda de un conjunto de entrenamiento pero, y aquí viene lo grave, también estamos entrenando los parámetros del algoritmo de aprendizaje utilizando el conjunto de prueba, que debería ser siempre independiente si queremos utilizarlo como estimación no sesgada de la calidad de los modelos obtenidos. Los parámetros del algoritmo de aprendizaje, habitualmente conocidos como hiperparámetros, obviamente tendremos que ajustarlos. Pero, para ello, debemos siempre emplear un conjunto de datos independiente del conjunto de prueba. Por tanto, el conjunto de datos disponible lo dividiremos en conjunto de entrenamiento (para construir modelos), conjunto de validación (para probar distintos parámetros del algoritmo de aprendizaje) y conjunto de prueba (para evaluar la calidad del modelo). Una descomposición razonable podría ser 60 % para el conjunto de entrenamiento, 20 % para el conjunto de validación y 20 % para el conjunto de prueba (o 60%/15%/25 %).

La descomposición del conjunto de datos en conjuntos de entrena-

Ajustar los parámetros del algoritmo de aprendizaje usando el conjunto de prueba es habitual, incluso, en publicaciones académicas y revistas científicas consideradas de primer nivel. En parte, se debe a la necesidad de publicar de los investigadores, lo que a menudo les obliga a centrar sus esfuerzos en conseguir mejoras nimias en el rendimiento de un método particular introduciendo pequeñas variantes, cualquier cosa con tal de poder sumar un *paper* más a su currículum investigador.

miento, validación y prueba también nos puede ayudar a detectar casos de sobreaprendizaje. Si llegado un momento, observamos que la tasa de error de nuestro modelo comienza a aumentar sobre el conjunto de validación, entonces es que ha comenzado el proceso de sobreaprendizaje y tal vez deberíamos parar (una estrategia conocida como *early stopping* en el entrenamiento de redes neuronales).

Muestreo sin reemplazo: Validación cruzada

La estimación del error mediante un conjunto de prueba independiente se puede hacer más fiable si repetimos el proceso con diferentes conjuntos de prueba. En cada prueba, podríamos seleccionar aleatoriamente el conjunto de datos de entrenamiento (con estratificación, posiblemente) y evaluar las tasas de error (o cualquier otra medida de nuestro interés) promediando los resultados obtenidos en cada experimento. Tal como hemos descrito el proceso [*repeated holdout*], los diferentes conjuntos de entrenamiento y de prueba se solaparían en los distintos experimentos. Además, siempre existiría la posibilidad de que algunos ejemplos no se usasen nunca como parte del conjunto de entrenamiento de un modelo (y desconocemos su posible impacto en la calidad de los modelos obtenidos). Para solventar ambas limitaciones, lo habitual es recurrir a la validación cruzada.

La validación cruzada crea y evalúa múltiples modelos de forma sistemática, cada uno de ellos sobre diferentes subconjuntos del conjunto de datos disponible. La validación cruzada de k iteraciones o k -CV [*k-fold Cross-Validation*] se realiza de la siguiente manera:

- En primer lugar, se divide aleatoriamente el conjunto de datos en k subconjuntos disjuntos del mismo tamaño.
- A continuación, para cada uno de los k subconjuntos se construye un modelo utilizando dicho subconjunto como conjunto de prueba y empleando los $k - 1$ restantes como conjunto de entrenamiento.

Los k subconjuntos son siempre de intersección vacía y suelen ser, más o menos, del mismo tamaño (si el tamaño de nuestro conjunto de datos no es múltiplo de k , no tenemos por qué tirar algunos ejemplos para forzar que los k conjuntos sean exactamente iguales).

Típicamente, se crea una partición del conjunto de datos en 10 subconjuntos (10-CV). ¿Por qué $k = 10$? Oficialmente, porque se han realizado numerosos experimentos que muestran que la mejor forma de obtener una buena estimación es una validación cruzada estratificada con $k=10$, aun cuando se disponga de capacidad de cálculo para aumentar el número de ‘pliegues’ [*folds*].⁵⁴ Extraoficialmente, sólo tiene que mirarse las manos. Por el mismo motivo, utilizamos el sistema decimal y no el binario.

⁵⁴ Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. En *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, 1995. ISBN 1-55860-363-8. URL <http://dl.acm.org/citation.cfm?id=1643031.1643047>

Al usar validación cruzada, en la iteración i , se usa el subconjunto i como conjunto de prueba y los $k - 1$ subconjuntos restantes como conjunto de entrenamiento. Al final de cada iteración, obtendremos una medida \hat{m}_i de la calidad del modelo obtenido, evaluada siempre sobre el conjunto de prueba correspondiente. Finalmente, realizamos una estimación \hat{m}_{cv} de la calidad del modelo promediando las medidas individuales:

$$\hat{m}_{cv} = \frac{1}{k} \sum_{i=1}^k \hat{m}_i$$

Se pueden diseñar distintas variantes de la validación cruzada:

- “Leave one out” (literalmente, dejar uno fuera)

Se realiza una validación cruzada con k particiones del conjunto de datos, donde k coincide con el número de ejemplos disponibles. Dado que se suele hacer referencia al número de ejemplos con n , esta variante de la validación cruzada aparece, en ocasiones, como n -CV.

El principal inconveniente de este método es que, si disponemos de n ejemplos, hay que construir n modelos, lo que puede resultar prohibitivo si nuestro conjunto de datos es grande. Por eso sólo se utiliza eventualmente, con conjuntos de datos bastante pequeños. Entre sus ventajas hay que destacar que no involucra muestreo aleatorio y hace el mejor uso posible de los datos disponibles para entrenar modelos y evaluarlos utilizando un conjunto independiente. En este caso, se deja un único ejemplo en el conjunto de prueba para evaluar la calidad de los modelos obtenidos. Obviamente, no admite estratificación...

- *Validación cruzada estratificada*

Las particiones se realizan intentando mantener en todas ellas la misma proporción de clases que aparece en el conjunto de datos completo. Usualmente, se eligen directamente los k subconjuntos de forma que todos ellos estén ya estratificados, por lo que también lo estarán los distintos conjuntos de entrenamiento y prueba empleados en la validación cruzada.

Para afinar aún más en las estimaciones conseguidas por validación cruzada, se puede ejecutar la validación cruzada de forma repetida. Por ejemplo, podríamos repetir 10 veces una validación cruzada 10-CV. Los 100 modelos construidos, 10 por cada una de las 10 validaciones cruzadas, nos servirán para obtener una estimación más precisa de la calidad del modelo (la repetición reduce la varianza de la estimación final).

La validación cruzada, obviamente, sólo proporcionará estimaciones fiables si el conjunto de datos es representativo para el problema que

estemos resolviendo. Además, los distintos subconjuntos deben seleccionarse para ser también representativos. Si en el problema que estemos estudiando, el sistema evoluciona con el tiempo (i.e. no es estacionario), el conjunto de datos de prueba utilizado podría incluir sesgos que afecten a la calidad de nuestras estimaciones. ¿Se fiaría de un sistema de predicción de cotizaciones en bolsa entrenado con datos de hace cinco o diez años? ¿Y de un sistema de diagnóstico entrenado sólo con muestras de un segmento de la población diferente al segmento al que usted pertenece?

Otra posible fuente de error en nuestras estimaciones es que algunos ejemplos usados en el entrenamiento de un modelo también acaben estando presentes en el conjunto de prueba con el que se evalúa el modelo. Algo que siempre puede pasar por la existencia de ‘gemelos’ [*twinning*] en el conjunto de datos: muestras idénticas o casi idénticas que están presentes en el conjunto de datos del que extraemos los subconjuntos de entrenamiento y prueba.

Muestreo con reemplazo: Bootstrapping

La validación cruzada utiliza técnicas de muestreo sin reemplazo: el mismo ejemplo, una vez seleccionado, no puede volver a utilizarse. Gracias a ello, no aparecen ejemplos duplicados en el conjunto de entrenamiento y en el de prueba (más allá de los inevitables ‘gemelos’).

El *bootstrapping* es una técnica alternativa que recurre a técnicas de muestreo con reemplazo; esto es, una vez que se escoge un ejemplo, se vuelve a dejar en el conjunto de datos y puede que se vuelva a escoger posteriormente.

El método *bootstrap* fue propuesto por Bradley Efron, de la Universidad de Stanford, en 1979.⁵⁵ Se muestrea un conjunto de datos con n ejemplos para formar un nuevo conjunto de datos de n ejemplos. Al realizar el muestreo con reemplazo, algunos de los ejemplos aparecerán más de una vez en la muestra seleccionada. Esa muestra, del mismo tamaño que el conjunto de datos original, es la que se utiliza para entrenar un modelo. ¿Cómo se evalúa ese modelo? Recurriendo a los ejemplos del conjunto de datos original que no han sido incluidos en el conjunto de datos de entrenamiento.

Este método también se conoce como *0.632-bootstrap*. Cuando se escoge una muestra, se escoge siempre con probabilidad $1/n$. La probabilidad de que una muestra no sea escogida será, obviamente, $1 - 1/n$. Como se construye un conjunto de entrenamiento con n muestras, el proceso hay que repetirlo n veces. Por tanto, la probabilidad de que una muestra no sea escogida para formar parte del conjunto de entrenamiento será $(1 - 1/n)^n \approx e^{-1} = 0.368$. En otras palabras, el 36.8% de las muestras no se escogerá nunca y pasará el conjunto de prueba. El 63.2% restante formará parte del conjunto de entrenamiento, de ahí el nombre.

Bootstrapping, literalmente, hace referencia a las correas o tiras que llevan algunas botas para que resulte más sencillo ponérselas. Algunas zapatillas de deporte también las llevan en su parte posterior, lo que algunos aprovechan para atarlas (o colgarlas de cables en una práctica absurda conocida como *shoefti*). En Informática, también se utiliza el término para describir el proceso por el cual un sistema activa otro sistema más complejo, como en el arranque de un sistema operativo o durante el desarrollo de un compilador escrito en el propio lenguaje que compila.

⁵⁵ Bradley Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1): 1–26, 1979. ISSN 0090-5364. DOI: 10.1214/aos/1176344552

¿Cómo se estima la calidad de un modelo obtenido con *bootstrapping*? Si usamos directamente la evaluación que nos proporcione el conjunto de prueba escogido, es probable que nuestra estimación sea demasiado pesimista. Más que nada, porque apenas hemos empleado el 63 % de los datos disponibles para construir el modelo. Por este motivo, las medidas derivadas del conjunto de prueba se combinan con las obtenidas directamente del conjunto de entrenamiento de la siguiente forma:

$$\hat{m}_{bootstrap} = 0.632 \hat{m}_{prueba} + 0.368 \hat{m}_{entrenamiento}$$

Observe que la medida derivada del conjunto de entrenamiento (p.ej. el error de resustitución) recibe menor peso que la medida obtenida del conjunto de prueba (que contiene los ejemplos que nunca se seleccionaron para formar parte del conjunto de entrenamiento del modelo).

Si queremos una estimación más fiable, podemos repetir el proceso varias veces, seleccionando distintos conjuntos de muestras, y promediar los resultados obtenidos en las distintas ejecuciones:

$$\hat{m}_{bootstrap} = \frac{1}{k} \sum_{i=1}^k (0.632 \hat{m}_{prueba_i} + 0.368 \hat{m}_{entrenamiento_i})$$

Aunque no se utiliza tan a menudo como la validación cruzada, puede resultar útil si necesitamos estimar el rendimiento de un modelo cuando sólo disponemos de un conjunto de datos muy pequeño. Si tenemos muy pocos datos disponibles para construir un modelo, *bootstrapping* nos ofrece un mecanismo para construir un conjunto de entrenamiento del mismo tamaño que nuestro conjunto de datos original y, además, evaluarlo con la ayuda de un conjunto de prueba independiente del conjunto de entrenamiento.

NOTA: El *bootstrapping* no resulta demasiado recomendable para la detección de anomalías ni, en general, para los problemas de clasificación poco balanceada, ya que los ejemplos de las clases poco frecuentes tienen una probabilidad pequeña de entrar en el conjunto de datos de entrenamiento.

Finalización del modelo

Los métodos vistos en la sección anterior sirven para estimar la calidad que nos ofrece un modelo construido utilizando técnicas de aprendizaje automático. El uso de un conjunto de prueba independiente del conjunto de entrenamiento, validación cruzada o *bootstrapping* nos permite obtener estimaciones más fiables de las métricas que utilicemos para evaluar modelos alternativos.

Ahora bien, una vez estimada la calidad del modelo que podemos obtener gracias a una técnica particular de aprendizaje, ¿por qué no aprovechar todos los datos disponibles para construir el modelo final si sabemos que, cuantos más datos utilicemos en su entrenamiento, mejor será el modelo en la práctica? Llegados a este punto, los modelos obtenidos para estimar la calidad del algoritmo de aprendizaje utilizado podemos

descartarlos y entrenar un nuevo modelo que aproveche todos los datos disponibles.

Con las estimaciones correctamente realizadas, podemos olvidarnos de las particiones del conjunto de datos que realizamos para obtener un conjunto de prueba independiente del conjunto de entrenamiento o para repetir k experimentos con muestras diferentes del conjunto de datos (sin reemplazo en el caso de la validación cruzada, con reemplazo en *bootstrapping*). Ya han cumplido su misión con éxito y no resultan necesarias.

El paso final del proceso, por tanto, consiste en construir un modelo final, que es el que utilizaremos en la práctica para realizar predicciones en el mundo real, una vez finalizada la fase de entrenamiento en la que utilizábamos conjuntos separados para aprender (entrenamiento), ajustar parámetros (validación) y estimar métricas (prueba). Ese modelo final lo entrenaremos utilizando la técnica que hayamos seleccionado en función de los resultados observados empíricamente. Ajustaremos sus parámetros usando los valores que mejores resultados nos hayan proporcionado. Y, por supuesto, emplearemos todos los datos disponibles. Eso es todo.

¿No podemos utilizar como modelo final un modelo construido sólo a partir del conjunto de entrenamiento? ¿O el mejor de los que construimos al realizar una validación cruzada? Podemos. Tal vez nos ahorremos algo de tiempo, al no tener que volver a entrenar un modelo desde cero, pero es bastante probable que un modelo entrenado con todos los datos disponibles funcione mejor que un modelo entrenado con el subconjunto utilizado para estimar su rendimiento.

Si construimos un nuevo modelo aprovechando todos los datos, ¿cómo sabemos que el modelo funcionará bien si no lo hemos evaluado? ¿No podría ser diferente su rendimiento con respecto a los utilizados anteriormente? Ambas preguntas, aunque legítimas, son precisamente las preguntas para las que se diseñaron los métodos de remuestreo como la validación cruzada. Al usar dichos métodos, estamos respondiéndolas. Los resultados de la validación cruzada describen cómo de bien funcionará el modelo final. No sólo eso, además de tener una estimación puntual de cómo de bueno es (o malo), los resultados también nos ofrecen información acerca de la variabilidad que podemos esperar (su varianza). Por eso era tan importante diseñar correctamente los experimentos de evaluación e incidimos tanto en evitar que nuestra estimación estuviese sesgada.

Descomposición del error en sesgo y varianza

Supongamos que ha entrenado un modelo pero su rendimiento no es el que esperaba obtener. Tiene que depurar el proceso de aprendizaje para detectar el problema. Su sistema, ¿sufre de ceguera o ve alucinaciones? Los términos técnicos para estas patologías son sesgo y varianza. Un reloj

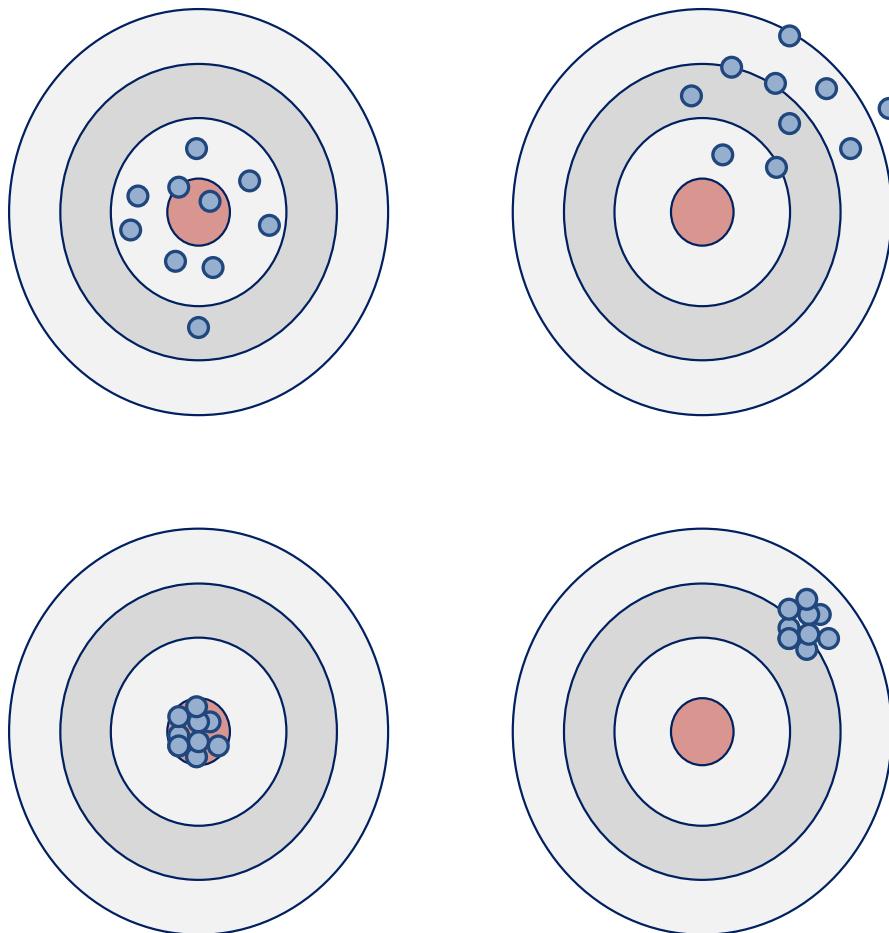


Figura 15: Representación gráfica del sesgo y de la varianza. Abajo a la izquierda: sesgo bajo, varianza reducida. Abajo a la derecha: sesgo elevado, varianza limitada. Arriba a la izquierda: sesgo acotado, varianza alta. Arriba a la derecha: sesgo y varianza prominentes.

atrasado, pero que por lo demás funciona bien, tiene un sesgo elevado pero una varianza reducida. Si el reloj se comporta de forma errática pero suele dar bien la hora, aunque se pase el día acelerando y frenando cual experimento relativista jugando con el tiempo, entonces tiene una varianza elevada pero un sesgo bajo.

Gráficamente, se puede interpretar el sesgo y la varianza con ayuda de una diana. Si juega a los dardos en un bar mientras bebe cerveza, algo que no recomendamos por la salud de los que le rodean, su varianza irá aumentando conforme suba su nivel de alcohol en sangre. Su sesgo, sin embargo, es algo que sólo podrá reducir con mucha práctica y algo de buen pulso. El alcohol no le ayudará a mejorar en eso, aunque tal vez sí a mejorar su percepción de lo que le rodea.

Matemáticamente, las discusiones sobre sesgo y varianza en aprendizaje automático provienen de un artículo de 1992 en el que se realiza una descomposición del error cometido por una red neuronal multicapa.⁵⁶

Supongamos que trabajamos en un problema de predicción numérica, como puede ser la regresión o la predicción de series temporales. Entonces,

¿Recuerda la definición de un problema según Weinberg? La diferencia entre las cosas como se perciben y como se deseán. El alcohol sólo actúa en la parte correspondiente a la percepción.

⁵⁶ Stuart Geman, Elie Bienenstock, y René Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, January 1992. ISSN 0899-7667. DOI: 10.1162/neco.1992.4.1.1

nuestro modelo realizará predicciones de la forma $\hat{y} = f(x)$. La salida del sistema \hat{y} sólo será una estimación del valor real y . El error cometido por el sistema será de la forma

$$\text{error}(x) = E[(\hat{y} - y)^2] = E[(f(x) - y)^2]$$

donde E representa el valor esperado para el error (por ejemplo, la media observada en un conjunto de datos de prueba).

Dado ese error, podemos estimar el sesgo o bias de nuestra estimación como sigue:

$$\text{bias}(x) = \text{sesgo}(x) = E[\hat{y} - y] = E[f(x) - y] = E(f(x)) - y$$

De forma similar, la varianza de nuestro estimador se puede definir como

$$\text{varianza}(x) = E[(f(x) - E(f(x)))^2]$$

Retomemos nuestra función de error y veamos cómo se puede descomponer ese error en términos del sesgo y de la varianza:

$$\begin{aligned} \text{error}(x) &= E[(f(x) - y)^2] \\ &= E[((f(x) - E(f(x))) + (E(f(x)) - y))^2] \\ &= E[(f(x) - E(f(x)))^2] \\ &\quad + E[(E(f(x)) - y)^2] \\ &\quad + 2E[(f(x) - E(f(x)))(E(f(x)) - y)] \\ &= E[(f(x) - E(f(x)))^2] \\ &\quad + E[f(x)]^2 - 2yE(f(x)) + y^2 \\ &\quad + 2E[f(x)E(f(x)) - yf(x) - E(f(x))^2 + yE(f(x))] \\ &= E[(f(x) - E(f(x)))^2] \\ &\quad + E[f(x)]^2 + y^2 - 2yE[f(x)] \\ &= E[(f(x) - E(f(x)))^2] \\ &\quad + (E(f(x)) - y)^2 \\ &= \text{varianza}(x) + \text{bias}^2(x) \end{aligned}$$

Esto es, el error cometido por nuestro modelo tiene dos componentes diferenciados que corresponden al sesgo y a la varianza del modelo aprendido. Si comprendemos las fuentes de error que ocasionan el sesgo y la varianza, podremos analizar los problemas de rendimiento de un algoritmo de aprendizaje. Una vez identificados esos problemas, tal vez podamos hacer algo para obtener mejores modelos. El sesgo y la varianza son, para el científico de datos, como el depurador para un programador.

Supongamos que hemos construido un modelo supervisado pero ese modelo, al evaluarlo en un conjunto de prueba independiente del conjunto

de entrenamiento (o al efectuar una validación cruzada), tiene una tasa de error que consideramos inaceptable para usarlo en la práctica. ¿Qué hacemos a continuación?

- *Errores debidos al sesgo [bias]*

El sesgo mide la desviación entre las predicciones de nuestro modelo y los valores correctos que tratamos de predecir. Nos indica si, sistemáticamente, estamos cometiendo errores siempre en el mismo sentido.

Si nuestro modelo se comporta peor de lo esperado y observamos que el error sobre el conjunto de entrenamiento es similar al error sobre el conjunto de prueba, entonces puede que este error se deba al sesgo de nuestro modelo.

Puede que nuestro modelo no sea lo suficientemente flexible para modelar los datos de nuestro conjunto de entrenamiento. Tal vez porque no resulta adecuado como modelo en sí (p.ej. intentar ajustar un conjunto de datos complejo con una simple regresión lineal). Tal vez porque estemos simplificando en exceso la complejidad del modelo, mediante algún mecanismo de regularización incluido en nuestro algoritmo. En ese caso, deberíamos reducir la regularización.

Puede que las características de los datos con los que estamos intentando realizar una predicción no sean suficientes para poder realizar una predicción correctamente. En tal caso, podemos intentar obtener más características y ampliar el número de atributos de nuestro conjunto de entrenamiento. Cuando el sesgo es la causa del error, conseguir más ejemplos de entrenamiento, por sí mismo, no nos ayudará demasiado. Pero sí ampliar el conjunto de características utilizado, incluso aumentándolo artificialmente (p.ej. incluyendo los productos de pares de características podríamos realizar un ajuste cuadrático de los datos usando un sencillo algoritmo de regresión lineal).

- *Errores debidos a la varianza*

La varianza mide la variabilidad del modelo obtenido en función de los datos concretos utilizados en cada momento. Esto es, si construimos distintos modelos del mismo tipo partiendo de diferentes conjuntos de entrenamiento, la varianza representa cuánto variarán las predicciones de los distintos modelos cuando los utilicemos para realizar una predicción concreta sobre un caso de prueba.

Si nuestro modelo se comporta peor de lo esperado y observamos que el error sobre el conjunto de entrenamiento es bajo y además es mucho menor que el error medido sobre el conjunto de prueba, entonces es probable que el error se deba a la varianza.

Cuando el error se debe a la varianza, la solución más sencilla pasa por aumentar el número de casos de nuestro conjunto de entrenamiento. Si esto no es posible por limitaciones temporales, técnicas o presupuestarias, también podemos reducir el conjunto de características utilizado para construir el modelo utilizando algún mecanismo de selección de características (recuerde: en ocasiones, menos es más). Si nuestro algoritmo de aprendizaje admite algún parámetro de regularización, puede que aumentar la regularización nos ayude a mejorar los resultados de nuestro modelo.

Aunque algunas técnicas de aprendizaje puedan obtener resultados espectaculares cuando disponemos de un conjunto de datos lo suficientemente grande, en ocasiones cometan errores garrafales que nos hacen dudar de su validez. Por ejemplo, una red neuronal empleada para reconocer objetos en imágenes podría decirnos que un cepillo de dientes es, en realidad, un bate de béisbol. Ese tipo de fallos pueden causar más de un quebradero de cabeza, como cuando el buscador de imágenes de Google llegó a clasificar imágenes de afroamericanos como gorilas y el agente conversacional Tay de Microsoft aprendió lo que no debía de Twitter. Otro sistema neuronal de Google llamado Word2Vec, que convierte palabras en vectores numéricos, se construyó utilizando millones de palabras de Google News y es capaz de establecer analogías políticamente incorrectas, del tipo “padre es a doctor como madre a enfermera”. Obviamente, los resultados obtenidos con un algoritmo de aprendizaje sólo pueden ser tan buenos como los datos de los que se extrajeron, que pueden incluir sesgos implícitos.

Utilicemos la técnica que utilicemos, siempre observaremos cierto error, independientemente de lo que hagamos. Aparte del sesgo y la varianza, existe un componente adicional del error que no puede eliminarse. Ese error irreducible es inherente al problema que pretendemos resolver y ningún modelo puede eliminarlo. Si dispusiésemos del modelo perfecto y de un conjunto de datos infinito para ajustar sus parámetros, tal vez podríamos reducir sesgo y varianza a cero. Pero en el mundo real trabajamos siempre con modelos imperfectos y conjuntos de datos finitos, lo que suele implicar un compromiso entre la varianza y el sesgo.

Intuitivamente, se podría pensar que se debe minimizar el sesgo aun a costa de que aumente la varianza. Si vemos el sesgo como algo básicamente erróneo en nuestro modelo, existe una inclinación natural a reconocer que la varianza, aunque también sea negativa, es algo aceptable si la predicción es, en media, correcta. Sin embargo, algunos tipos de sesgo pueden cancelarse reduciendo la varianza.⁵⁷ En ocasiones, eso permite que algoritmos sencillos de aprendizaje automático obtengan resultados tan buenos como los que se pueden conseguir con técnicas mucho más complejas. También explica por qué, a menudo, combinar múltiples mode-

⁵⁷ Jerome H. Friedman. On bias, variance, 0/1-loss, and the curse-of-dimensionality. *Data Mining and Knowledge Discovery*, 1(1): 55–77, 1997. ISSN 1384-5810. doi: 10.1023/A:1009778005914

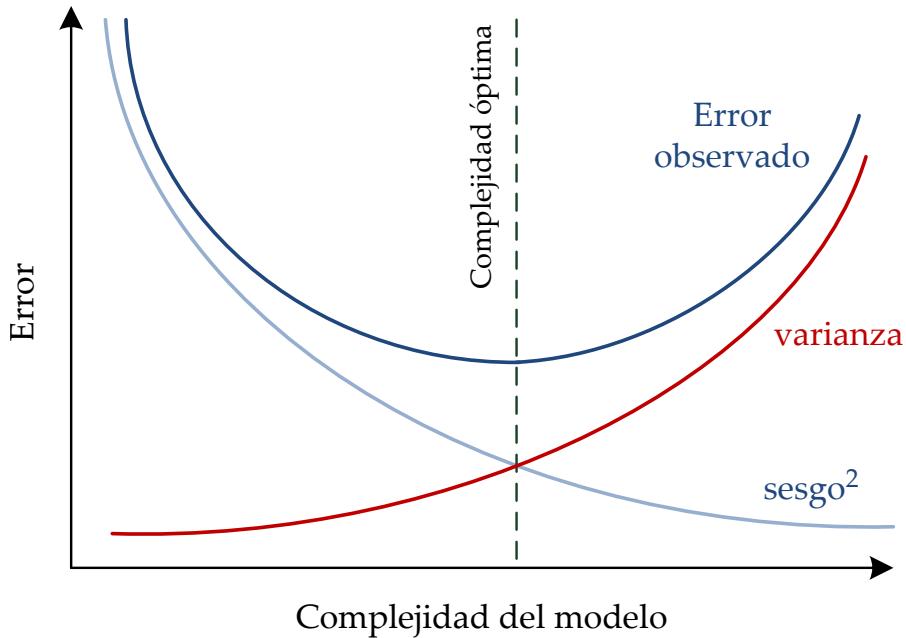


Figura 16: Descomposición del error en sesgo y varianza.

los en “ensembles” puede ayudarnos a mejorar los resultados individuales de un modelo particular.

¿Por qué resulta tan poco intuitivo todo esto? Una varianza elevada y un sesgo bajo funcionan bien estadísticamente. En media, las predicciones son correctas y unos errores se compensan con otros. Sin embargo, en la práctica disponemos de un único conjunto de datos. Lo importante es el rendimiento que le seamos capaces de extraer a ese conjunto de datos particular. Sesgo y varianza son igual de importantes para nosotros y no deberíamos focalizar nuestros esfuerzos en reducir uno a costa del otro.

El sesgo y la varianza también nos pueden ayudar a comprender el fenómeno del sobreaprendizaje. Un modelo con un elevado sesgo presenta un error elevado en el conjunto de entrenamiento, similar al error sobre el conjunto de prueba (o al medido mediante validación cruzada). Estamos en el lado izquierdo de la curva del error en función de la complejidad del modelo [*underfitting*]. Nuestro modelo no tiene un número suficiente de parámetros como para aprender correctamente todos los matices asociados a nuestro conjunto de datos. Un modelo con excesiva varianza es capaz de ajustar muy bien el conjunto de entrenamiento pero no funciona tan bien con el conjunto de prueba [*overfitting*]. Estamos en el lado derecho de la curva. Nuestro modelo tal vez tenga tantos parámetros ajustables, que no disponemos de los datos suficientes para poder estimar adecuadamente cuáles deberían ser sus valores correctos.

Conforme aumenta la complejidad del modelo, su sesgo se reduce y su varianza aumenta. Cuantos más parámetros se añaden a un modelo, su complejidad aumenta y la varianza se convierte en nuestra principal preo-

En las publicaciones académicas, se discuten propiedades formales asociadas a las técnicas de aprendizaje. Cuando el tamaño de nuestro conjunto de entrenamiento tiende a infinito (de ahí lo de académico), el sesgo del modelo desaparecerá si la técnica es asintóticamente consistente y la varianza del modelo no será peor que la de cualquier otro modelo potencial si la técnica es asintóticamente eficiente. En el mundo real, no tenemos muestras infinitas, por lo que algoritmos eficientes y consistentes asintóticamente tal vez funcionen peor que técnicas que carecen de esas propiedades formales.

cupación, ya que el sesgo disminuye progresivamente. Matemáticamente, el sesgo tiene una derivada negativa con respecto a la complejidad del modelo, mientras que la varianza tiene una pendiente positiva. Lo realmente importante es el error en su conjunto, no su descomposición particular, por lo que el punto justo de nuestro modelo será el correspondiente a un nivel de complejidad tal que un aumento en el sesgo sea equivalente a una reducción de la varianza. Formalmente, se podría expresar algo así:

$$\frac{d \text{ bias}^2}{d \text{ complejidad}} = -\frac{d \text{ varianza}}{d \text{ complejidad}}$$

Si nos pasamos de este punto, estaremos sobreajustando nuestro modelo [*overfitting*]. Si no llegamos, nuestro modelo no dispone de la complejidad suficiente para representar la riqueza de nuestro conjunto de datos [*underfitting*].

Se trata sólo de una ficción. En la práctica, no disponemos de mecanismos que nos permitan determinar este punto ideal desde un punto de vista analítico. Tendremos que explorar varias opciones y quedarnos con la que mejor funcione, de acuerdo con la métrica que cuidadosamente hayamos escogido previamente.

Técnicas de aprendizaje

Se suele decir que no se puede dominar lo que no se conoce, por lo que nos vendrá bien familiarizarnos con las diferentes técnicas de aprendizaje automático que existen. En función del contexto, y de la disciplina científica de la que provengamos, las técnicas de aprendizaje automático han recibido diferentes nombres: reconocimiento de formas (o patrones), modelado estadístico, análisis de datos, minería de datos o data mining, KDD o knowledge discovery, analítica predictiva [predictive analytics], sistemas adaptativos o, incluso, sistemas auto-organizativos.

También se suele decir que uno no comprende realmente algo hasta que es capaz de explicárselo a otra persona. O, yendo un poco más allá, hasta que es capaz de expresarlo en forma de algoritmo. El físico Richard Feynman afirmaba no comprender aquello que no podía crear. Los físicos, como los ingenieros, trabajan con fórmulas y ecuaciones. Éstas, en realidad, no son más que tipos muy particulares de algoritmos.

Por los teoremas de Wolpert y Kleinberg, ya sabemos que ninguna técnica particular de aprendizaje automático, ni supervisada ni no supervisada, resultará la mejor para todo problema que se nos pueda presentar en la práctica. Como sucede en otras disciplinas, se han desarrollado diferentes escuelas de pensamiento en aprendizaje automático. Cada una de ellas aborda el problema del aprendizaje desde una perspectiva diferente. Para analizarlas, utilizaremos la clasificación propuesta por Pedro Domingos, de la Universidad de Washington en Seattle.⁵⁸ Domingos distingue cinco grandes familias que se han formado en torno a la Inteligencia Artificial:

- Los *simbólicos* centran su atención en la interpretación filosófica, lógica y psicológica del aprendizaje: un proceso de inducción que no es más que una deducción a la inversa.
- Los *analógicos* basan el aprendizaje en la extrapolación a partir de ejemplos conocidos, mediante la realización de juicios de similitud. Su razonamiento por analogía tiene bases psicológicas y, en ocasiones, se acaba traduciendo en un problema matemático de optimización, como sucede con las máquinas de vectores de soporte [SVM].
- Los *bayesianos* utilizan técnicas estadísticas de inferencia probabilística, basadas en última instancia en el teorema de Bayes, como sucede con las redes bayesianas.
- Los *evolutivos* extraen sus conclusiones de la teoría de la evolución de Darwin y de las bases de la genética, de donde se inspiran para

⁵⁸ Pedro Domingos. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World.* Basic Books, 1st edition, 2015. ISBN 0465065708

desarrollar técnicas como los algoritmos genéticos.

- Los *conexionistas* se inspiran en el cerebro humano, intentando hacer ingeniería inversa de su funcionamiento con la ayuda de físicos y neurocientíficos (aunque sus modelos no siempre acaben siendo biológicamente plausibles).

Cada una de las comunidades anteriores resuelve problemas similares utilizando diferentes métodos para llegar a una solución. Como también sucede en otras disciplinas académicas,⁵⁹ la formación de escuelas, en ocasiones, deriva en la configuración de sectas. Inspirado en un documental sobre la Iglesia de la Cienciología, el economista Paul Romer critica las características que adquieren las escuelas académicas: enorme autoconfianza; comunidad monolítica; sentido de identificación con la religión o la política; frontera muy definida entre el grupo propio y los demás (lo que les lleva a minusvalorar, despreciar y atacar a los de fuera); sesgo optimista respecto de las pruebas que confirman sus hipótesis; y, por último, incapacidad para evaluar el riesgo asociado a su programa de investigación. Las escuelas académicas se convierten, pues, en bandos enfrentados que compiten por acaparar la atención, ya sea la financiación de proyectos y programas de investigación o la primacía intelectual entre el público general. Algo que desde fuera podría verse como un simple mecanismo freudiano de transferencia, mediante el cual vuelcan sobre otros grupos sus características psicológicas y frustraciones propias, previamente reprimidas, este tipo de enfrentamientos puede tener resultados nefastos en el desarrollo de una disciplina.

No en vano, como reconocía el físico Werner Heisenberg, los desarrollos más provechosos han surgido siempre donde se encontraron dos formas de pensar diferentes. Afortunadamente, muchos libros de texto^{60,61} se olvidan de las rencillas internas entre comunidades de especialistas. Intentan eliminar las fronteras que delimitan los distintos paradigmas existentes, que son artificiales en muchas ocasiones. Cuando ofrecen una perspectiva integradora de un área, como sucede con la tabla periódica de los elementos en Química, permiten que profanos en el tema, ajenos a las rivalidades entre subáreas de conocimiento, puedan aportar su perspectiva fresca y realizar contribuciones significativas. En el caso del aprendizaje automático, nuestra tabla periódica de métodos de aprendizaje debería incluir, al menos, técnicas de inducción de reglas, aprendizaje basado en casos, métodos probabilísticos, algoritmos evolutivos y redes neuronales artificiales.

Ya que ninguna de las técnicas particulares es perfecta, el desafío final sería construir un sistema capaz de aprender todo lo que se puede aprender a partir de los datos. El “algoritmo maestro”, según Pedro Domingos, capaz de lograr todo lo que el cerebro humano consigue hacer, la evolución ha creado y el conocimiento científico ha acumulado. Pero

⁵⁹ Paul Romer. *The trouble with macroeconomics*. *The American Economist*, Commons Memorial Lecture of the Omicron Delta Epsilon Society, 2016. URL <https://paulromer.net/wp-content/uploads/2016/09/WP-Trouble.pdf>

⁶⁰ Pat Langley. *Elements of Machine Learning*. Morgan Kaufmann Publishers, 1995. ISBN 1558603018

⁶¹ Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, 1st edition, 1997. ISBN 0070428077

no nos desviemos demasiado del tema y entremos en un terreno tan resbaladizo.

Las familias de técnicas asociadas a las diferentes escuelas ya son, en cierto modo, universales. Al menos, en el siguiente sentido: si les proporcionamos una cantidad suficiente de datos, pueden aproximar cualquier función con el grado de precisión que sea necesario. Es la forma matemática de decir que son capaces de aprender cualquier cosa.

El problema es que nuestro tiempo es limitado y la cantidad de datos 'suficiente' podría muy bien ser infinita. Entonces, no nos queda más remedio que realizar suposiciones que simplifiquen nuestro problema en la práctica. Y diferentes técnicas realizan distintas suposiciones. Veamos en qué consisten con un poco más de detalle.

Técnicas simbólicas: I.A. simbólica

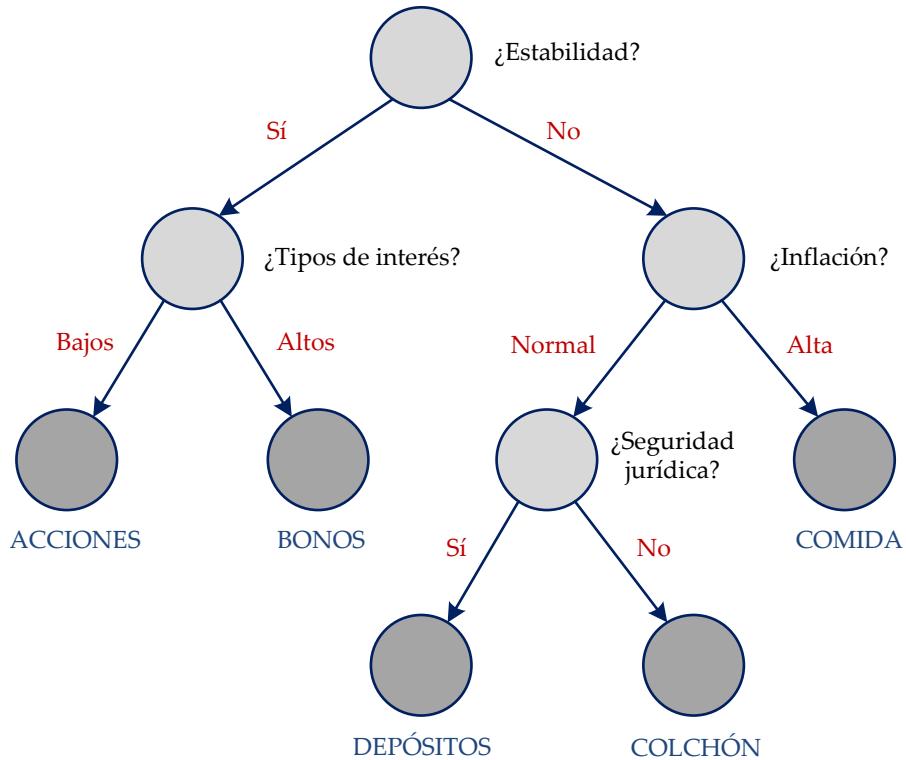
Las técnicas simbólicas de aprendizaje automático hacen énfasis en la representación simbólica del conocimiento adquirido por inducción (el proceso de deducción a la inversa que forma la base del método científico). La I.A. simbólica fue la corriente principal desde los orígenes de la Inteligencia Artificial en Informática y de las ciencias cognitivas en Psicología. De ahí que John Haugeland hiciera referencia a ella con el término GOFAI [*Good Old-Fashioned Artificial Intelligence*].

Desde el punto de vista formal, un símbolo no es más que la representación de una idea. Un símbolo asocia un signo lingüístico a un concepto. El signo lingüístico, que puede ser una secuencia de fonemas o de grafemas, es la representación del símbolo y se denomina significante [signifier]. El concepto asociado al símbolo es su significado [signified], el contenido del símbolo. Desde el punto de vista semiótico, el tercer componente asociado a un signo es su referente, el objeto real al que alude el signo.

Los simbólicos interpretan los problemas desde un punto de vista abstracto y, a la hora de resolver un problema, intentan incorporar el conocimiento ya existente acerca del mismo. La mayoría de los sistemas expertos, por ejemplo, utilizan este enfoque. Mediante la representación del conocimiento en forma de reglas de tipo IF-THEN, esos sistemas expertos intentan capturar el conocimiento de un experto humano de forma simbólica.

Algunos de los investigadores más destacados dentro del área del aprendizaje automático de tipo simbólico son Tom Mitchell (Carnegie Mellon University, Pittsburgh, Pennsylvania), Ryszard S. Michalski (Universidad de Illinois en Urbana-Champaign, Illinois, y George Mason University, en Fairfax, Virginia), Oren Etzioni (Universidad de Washington y Allen Institute for Artificial Intelligence, Seattle, Washington), Stephen Muggleton (Imperial College, Londres) o John Ross Quinlan (fundador de RuleRequest, Sydney, Australia).

Dado que algunos símbolos carecen de referente, o es muy difícil identificarlo, algunos lingüistas excluyen el referente de la definición de signo lingüístico, por lo que un símbolo queda reducido a significante y significado.



Árboles de decisión

Existen innumerables algoritmos de aprendizaje simbólico. Entre ellos se encuentran los más conocidos: los algoritmos de inducción de árboles de decisión, clasificación o identificación. Los árboles de decisión son, tal vez, el modelo de aprendizaje supervisado más utilizado. Su principal virtud radica en su interpretabilidad: es muy fácil interpretarlos, al menos cuando son pequeños, ya que cada camino desde la raíz del árbol hasta una de sus hojas equivale a una regla de tipo IF-THEN. Por su propia topología, son extremadamente eficientes para clasificar nuevos datos: basta con dejar caer cada ejemplo desde la raíz del árbol, siguiendo las ramas correspondientes a los valores de los atributos del ejemplo que queremos clasificar, hasta llegar a una hoja del árbol, la cual nos indicará la clase del ejemplo. Además, los árboles de decisión ofrecen una precisión comparable a otras técnicas de aprendizaje, por lo que pueden ser una buena alternativa siempre que uno quiera obtener un modelo de clasificación interpretable: la decisión del clasificador se justifica, simplemente, con la regla IF-THEN asociada al camino que nos llevó desde la raíz del árbol hasta una de sus hojas.

Ahora bien, ¿cómo se construyen los árboles de decisión? No podemos enumerar todos los árboles de decisión que se podrían construir a partir de un conjunto de datos para después compararlos (es un problema

Figura 17: Árbol de decisión para decidir cómo invertir nuestros ahorros en función de diversas variables.

NP). Para obtener, de forma eficiente, un árbol de clasificación que explique nuestros datos del conjunto de entrenamiento, podemos diseñar un algoritmo *greedy* (voraz o, incluso, glotón en algunas traducciones demasiado literales del inglés).

Emplearemos una estrategia de tipo ‘divide y vencerás’ para construir el árbol. Inicialmente, todos los ejemplos de entrenamiento se sitúan en la raíz de lo que será nuestro árbol de decisión. Ese conjunto se irá dividiendo recursivamente en función del atributo que se seleccione para ramificar el árbol en cada uno de sus nodos. Cuando, llegados a un nodo, todos los ejemplos que queden pertenezcan a la misma clase, podemos detener el proceso de ramificación: se añade una hoja al árbol con la etiqueta de la clase de los ejemplos correspondientes a ese nodo. Dada la estrategia utilizada en la construcción del árbol, de forma descendente desde la raíz del árbol, se suele hacer referencia a los algoritmos de inducción de árboles de decisión mediante el acrónimo TDIDT [*Top-Down Induction of Decision Trees*].

¿Cómo se decide en cada momento la forma de ramificar el árbol de decisión? Siguiendo el principio filosófico de la navaja de Occam, nuestro objetivo será obtener un árbol relativamente pequeño que sea capaz de explicar la clase asociada a los ejemplos de nuestro conjunto de entrenamiento. Para ello, utilizaremos un criterio heurístico para seleccionar qué atributo utilizamos para ramificar localmente el árbol. Esos criterios heurísticos se denominan reglas de división y se diseñan para intentar separar unas clases de otras. Algunos están basados en Teoría de la Información, como los criterios de ganancia de información utilizados por Quinlan en sus algoritmos ID3 y C4.5, que han dado lugar a implementaciones como el C5.0 de RuleQuest o el J4.8 incluido en Weka. Otra regla de división habitual es el índice de Gini, que los economistas utilizan para medir la desigualdad. Es la utilizada en los árboles de clasificación y regresión [CART: Classification And Regression Trees] propuestos por Leo Breiman y sus colaboradores de la Universidad de California en Berkeley. También es la empleada por los algoritmos SLIQ [Supervised Learning In Quest] y SPRINT [Scalable PaRallelizable INduction of decision Trees] que se desarrollaron en el seno del proyecto Quest de IBM. Existen otras muchas reglas de división. Algunas tienen alguna justificación formal desde el punto de vista estadístico (p.ej. χ^2 [chi-cuadrado]) o desde el punto de vista de la Teoría de la Información (p.ej. MDL [*Minimum Description Length*]). Otras son meramente heurísticas. Pero ninguna de ellas es significativamente mejor que todas las demás,⁶² lo que no debería extrañarnos demasiado a estas alturas (¿recuerda el teorema de Wolpert?).

Además de la regla de división, existen otros parámetros que influyen en la construcción de un árbol de decisión. Por ejemplo, su topología: CART siempre construye árboles binarios, mientras que algoritmos como

La navaja de Occam se atribuye a un monje franciscano inglés del siglo XIV, Guillermo de Ockham, una villa del condado de Surrey (deletreado Occam en latín). Este principio, establece que, dadas varias hipótesis alternativas para explicar un fenómeno, preferiremos la explicación más sencilla posible.

⁶² Fernando Berzal, Juan Carlos Cubero, Fernando Cuenca, y María José Martín-Bautista. On the quest for easy-to-understand splitting rules. *Data and Knowledge Engineering*, 44 (1):31–48, 2003. DOI: 10.1016/S0169-023X(02)00062-9

C4.5 emplean ramificaciones binarias sólo para los atributos continuos y construyen árboles n-arios para los atributos de tipo categórico. Algunos algoritmos ni siquiera son capaces de trabajar directamente con atributos de tipo continuo, lo que hace necesario un proceso previo de discretización, que convierta los atributos numéricos en atributos categóricos.

Otra característica que puede variar de un algoritmo a otro es qué hacen cuando nos encontramos valores desconocidos en nuestro conjunto de entrenamiento. Algunos métodos, llegados a un nodo en el que se ramifica el árbol utilizando un atributo que tiene un valor nulo para el ejemplo que queremos clasificar, son capaces de seguir por varias ramas simultáneamente y combinar los resultados de la clasificación del ejemplo en los diferentes subárboles del nodo en el que nos habíamos atascado. Otros algoritmos, simplemente, no están preparados para tratar con valores nulos y requieren que, previamente, los rellenemos de alguna forma (utilizando algún método de imputación de valores).

Otro problema habitual es que el árbol construido a partir de un conjunto de entrenamiento sea excesivamente complejo. Si exigimos que las hojas del árbol sólo incluyan ejemplos de una misma clase, eso puede ocasionar que nuestro árbol de decisión incluya nodos debidos a las peculiaridades del conjunto de entrenamiento utilizado, el cual puede incluir ruido y errores. Esos nodos, habitualmente, no nos serán útiles para generalizar correctamente, por lo que haríamos bien en no incluirlos en el árbol de decisión final. Para prevenir este sobreaprendizaje, podemos detener la construcción del árbol antes de que todos los ejemplos sean de la misma clase. Simplemente, añadimos una hoja etiquetada con la clase más frecuente cuando pensamos que no tiene mucho sentido seguir ramificando el árbol. Otra posibilidad es, tras construir un árbol de decisión completo, podarlo de alguna forma. Las técnicas de poda pueden eliminar ramas del árbol de decisión, bien utilizando un conjunto de datos distinto al conjunto de entrenamiento (como sucede en la poda por coste-complejidad de CART), o bien directamente, realizando una estimación estadística del error que podría introducir sustituir un subárbol completo por una simple hoja (como hace la poda pesimista de C4.5). La poda, obviamente, introducirá errores de clasificación adicionales en el conjunto de entrenamiento. Sin embargo, realizada correctamente, aumentará la capacidad de generalización del árbol para datos diferentes a los del conjunto de entrenamiento, que es lo que realmente nos interesa.

Dadas las ventajas que ofrecen los árboles de decisión como modelo de clasificación, entre las que destacan su interpretabilidad y la eficiencia de los algoritmos que los construyen, no es de extrañar que se hayan propuesto numerosas técnicas que hacen más eficiente y escalable el proceso de inducción de árboles de decisión. PUBLIC integra la poda en el proceso de construcción del árbol, en vez de realizarla a posteriori. RainForest aísla lo que determina la escalabilidad de un algoritmo TDIDT

para que se pueda utilizar con conjuntos de entrenamiento enormes. Finalmente, BOAT lleva esa estrategia hasta el extremo y sólo necesita recorrer secuencialmente dos veces el conjunto de datos de entrenamiento para construir el árbol de decisión completo.

Inducción de reglas y listas de decisión

Pese a la atención que han acaparado tradicionalmente, los árboles de decisión no son, ni mucho menos, la única forma de construir modelos de clasificación basados en reglas de tipo simbólico. Obviamente, se pueden derivar conjuntos de reglas de un árbol de decisión (cada camino desde la raíz hasta una hoja da lugar a una regla). Pero también se han diseñado algoritmos específicos para la inducción de reglas, los cuales permiten construir conjuntos de reglas que luego podemos utilizar para clasificar.

Desde los años 80 se interpretó la generalización como un proceso de búsqueda de reglas que encajasen con nuestros datos de entrenamiento. Es el caso de los espacios de versiones de Mitchell o de la metodología STAR de Michalski, una forma de aprender descripciones ‘estructurales’ a partir de ejemplos que se empleó en sistemas como INDUCE o AQ. En los años 90, Quinlan propuso su algoritmo FOIL [*First Order Inductive Learner*] y Muggleton acuñó el término programación lógica inductiva o ILP por sus iniciales en inglés [*Inductive Logic Programming*].

Con la llegada de las técnicas de minería de datos, se refinaron múltiples variantes de algoritmos, más o menos eficientes, que eran capaces de extraer reglas directamente a partir de un conjunto de entrenamiento.

¿Cómo se aprende una regla directamente a partir de un conjunto de datos? Generalmente, se empieza con la regla más general posible, aquella que asigna una misma clase a cualquier ejemplo (un clasificador por defecto). A continuación, se le van añadiendo antecedentes a la regla para maximizar su “calidad”, que puede evaluarse combinando su capacidad de cobertura (cuántos ejemplos cubre en el conjunto de entrenamiento) y su precisión (cuántos ejemplos es capaz de clasificar correctamente). Diferentes algoritmos emplearán distintos métodos heurísticos para ir descubriendo reglas individuales.

Los algoritmos de inducción de reglas más conocidos tal vez sean los que inducen listas de decisión. Una lista de decisión es una lista ordenada de reglas. Para clasificar un ejemplo dado, se van considerando las reglas en orden hasta que una de ellas sea aplicable a dicho ejemplo. Esa última regla será la que se emplee para clasificar el ejemplo. A diferencia de los algoritmos de inducción de árboles de decisión, que utilizan una estrategia ‘divide y vencerás’, los algoritmos de inducción de listas de decisión emplean una estrategia ‘separa y vencerás’. Las reglas de la lista se aprenden de una en una. Cada vez que se añade una regla a la lista de decisión, se eliminan del conjunto de entrenamiento todos los casos

cubiertos por la regla seleccionada. El proceso se repite iterativamente hasta que se cumpla alguna condición de parada preestablecida, momento en el que cerraremos la lista de decisión con una regla por defecto (la que clasificará todos los ejemplos no cubiertos por ninguna de las reglas de la lista de decisión). Algunos ejemplos conocidos de algoritmos de inducción de listas de decisión, aparte del ya mencionado FOIL, son los algoritmos CN2 de Clark y Boswell, el método RIPPER de Cohen o la estrategia PNrule de Agarwal y Joshi.

Estos algoritmos, no obstante, no son tan eficientes y escalables como los que construyen árboles de decisión, entre otros motivos por su carácter secuencial. A cambio, las técnicas de inducción de reglas ofrecen un mayor grado de flexibilidad a la hora de construir modelos de clasificación compactos (modelos más simples, o bien con menos reglas de clasificación, o bien con reglas más sencillas). Por desgracia, suelen ser demasiado ineficientes para poder aplicarse con éxito en la resolución de problemas de minería de datos.

Este hecho hizo que se propusiese el uso de técnicas más eficientes de extracción de reglas para la construcción de clasificadores simbólicos. A partir de reglas de asociación, una técnica no supervisada muy habitual en minería de datos, se pueden construir clasificadores asociativos. Los clasificadores asociativos permiten la construcción de modelos de clasificación simples, inteligibles y robustos de forma eficiente y escalable. Para ello, pueden recurrir a diferentes estrategias, desde aquéllos que utilizan reglas de asociación, tal cual, hasta los que emplean las reglas de asociación para construir híbridos de árboles y listas de decisión. Los primeros han de establecer formas de resolver conflictos cuando varias reglas se solapan y proporcionan clasificaciones contradictorias. Los segundos imponen una estructura al modelo de clasificación para evitar que esto pueda llegar a suceder, p.ej. ART [*Association Rule Tree*].

Técnicas analógicas: Reconocimiento de patrones

La segunda familia de técnicas de aprendizaje automático (la quinta tribu para Pedro Domingos) es la formada por un ecléctico grupo que suele hacer más énfasis en la formulación matemática que en el aspecto simbólico del aprendizaje. A falta de un nombre mejor, nos quedaremos con la etiqueta que les asocia Domingos: los analógicos [*analogizers*].

En vez de recurrir a la lógica simbólica, los analógicos extrapolan a partir de los datos conocidos utilizando juicios de similitud que les permiten establecer analogías. Si parece un pato, se mueve como un pato y grazna como un pato... entonces puede que sea un pato. Es la llamada prueba del pato, una visión informal del razonamiento abductivo. Otra forma de razonamiento más, junto a la deducción y la inducción. La abducción llega a una hipótesis a partir de la descripción de los

ART construye un árbol de decisión en el que se permite la utilización simultánea de varios atributos y se agrupan en una rama ‘else’ las ramas del árbol menos interesantes a la hora de clasificar directamente los datos. Para no descuidar la eficiencia del proceso de aprendizaje, ART emplea técnicas de extracción de reglas de asociación para formular hipótesis complejas y métodos de discretización para trabajar en dominios continuos de forma eficiente.

hechos observados. Esta hipótesis es una conjetura que explica las posibles razones del hecho observado, que se convierte en un caso particular de la regla propuesta. La deducción nos permite derivar las consecuencias lógicas de algo: los hombres son mortales y Sócrates es un hombre, luego Sócrates es mortal. La inducción, inferir algo posible a partir de los hechos observados, aunque no sea necesariamente cierto: todos los cisnes que hemos observado son blancos, luego todos los cisnes son blancos (ignorando la posibilidad de un cisne negro). La abducción, por su parte, infiere una causa a partir de la consecuencia observada: si vemos algo moverse, asumimos que alguna fuerza externa inició su movimiento.

Desde el punto de vista lógico, la abducción corresponde a la falacia de afirmar el consecuente o falacia *post hoc ergo propter hoc*; “después de esto, luego a consecuencia de esto”] porque ignora la posibilidad de que existan otras causas que estemos ignorando y expliquen nuestras observaciones (igual la fuerza que inició el movimiento no fuese externa, después de todo). Desde el punto de vista lógico, si $p \rightarrow q$, podemos inferir que q es cierto si observamos p , o bien $\neg p$ dado $\neg q$ ($p \rightarrow q \leftrightarrow \neg p \vee q \leftrightarrow \neg q \rightarrow \neg p$), pero nunca que p sea cierto porque observemos q . La falacia *post hoc* confunde correlación con causalidad. El gallo siempre canta antes de que salga el Sol; por tanto, el gallo hace que salga el Sol. Ingenuo Persuasible visitó al homeópata y, posteriormente, se curó de su enfermedad; por tanto, la homeopatía cura. Evidentemente, correlación no implica causalidad (aunque nos pueda dar alguna pista en ocasiones).

Las técnicas de aprendizaje incluidas en esta familia emplean razonamientos de tipo analógico: si este caso es similar a este otro que observamos con anterioridad, entonces es posible que se comporte igual que el ya observado. A diferencia de la abducción, que pretende inferir la mejor explicación genérica a partir de un hecho observado particular, la analogía es menos ambiciosa y se limita a razonar sobre casos particulares, de particular a particular. Haciendo juicios de similitud, se diseñan métodos algorítmicos y numéricos para construir modelos predictivos.

La similitud, como ya discutimos al analizar las técnicas no supervisadas de agrupamiento, siempre será subjetiva. Dependerá de la perspectiva desde la que veamos los datos. De ahí que los métodos analógicos combinen, en cierto modo, psicología (similitud subjetiva) y matemáticas (optimización).

La separación entre las técnicas simbólicas de la I.A. clásica y las técnicas analógicas proviene del énfasis, casi obsesivo, en intentar comprender las bases de la inteligencia humana por parte de los simbólicos. Los analógicos, más pragmáticos, se centraron en el desarrollo de técnicas algorítmicas que permitiesen resolver problemas prácticos de reconocimiento de formas [*pattern recognition*] o reconocimiento de patrones si hacemos una traducción más literal del inglés.

Peter Hart, Vladimir Vapnik y Douglas Hofstadter son algunos de

los nombres más conocidos que podríamos adscribir a la familia de los analógicos. Peter Hart es conocido por sus aportaciones a la Inteligencia Artificial y a la robótica como investigador del SRI [Stanford Research Institute], su libro de texto⁶³ y ser fundador de Ricoh Innovations (no confundir con la marca japonesa de impresoras). Vladimir Vapnik, matemático ruso que trabajó en los legendarios Laboratorios Bell de la AT&T en Holmdel, New Jersey, fue el coinventor de las máquinas de vectores de soporte SVM [*Support Vector Machines*], tras lo que pasó por NEC, la Universidad de Columbia en Nueva York y Facebook. Douglas Hofstadter, de la Universidad de Indiana, es hijo del Nobel de Física Robert Hofstadter y recibió el premio Pulitzer en 1979 por su obra “*Gödel, Escher, Bach*”.⁶⁴ Este último, no obstante, ha manifestado públicamente no tener verdadero interés por los ordenadores y se muestra escéptico con muchos de los proyectos relacionados con Inteligencia Artificial. Al fin y al cabo, que Deep Blue venciese a Kasparov tiene poco que ver con que los ordenadores se doten de verdadera inteligencia, por mucha publicidad que generase el acontecimiento.

¿Cuáles son las técnicas de aprendizaje automático que podríamos adscribir a la familia de los analógicos? Desde luego, todas las que utilizan los vecinos más cercanos, buscando casos similares para establecer analogías. Entre ellas, las utilizadas por los sistemas de recomendación de Amazon o Netflix, por ejemplo. Aunque otros quizás las vean como un tipo muy particular de red neuronal, las máquinas de vectores de soporte también son analógicas. Los ejemplos más cercanos a la frontera de decisión son los denominados vectores de soporte, de ahí el nombre de la técnica.

Un caso paradigmático de la utilidad de este tipo de técnicas es la historia de John Snow, el médico londinense que identificó el origen de un brote de cólera en 1854, un siglo antes de que se formalizasen los clasificadores basados en los k vecinos más cercanos, k -NN [*k Nearest Neighbors*]. La teoría que en esos momentos se barajaba consideraba que el “mal aire” londinense era la causa del cólera. Snow señaló en un mapa los casos conocidos de cólera, todos cerca de Broad Street en el distrito del Soho. En una prisión cercana, con cientos de reclusos, apenas había casos de cólera. Si el aire fuese la causa, esto no sería así, por lo que Snow atribuyó el cólera a una fuente de agua de la que bebían todos los afectados. Cerrada esa fuente, el brote de cólera remitió.

Vecinos más cercanos

Los clasificadores basados en casos, o clasificadores basados en instancias, son algo perezosos. Tanto, que es el nombre que reciben en inglés: *lazy learners*. Perezosos en el sentido de que no construyen un modelo de clasificación, sino que utilizan directamente los datos de su conjunto de

⁶³ Richard O. Duda, Peter E. Hart, y David G. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000. ISBN 0471056693

⁶⁴ Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979. ISBN 0465026850
A alguno le sonará el apellido por Leonard, uno de los personajes de la sitcom *La teoría del Big Bang* [*The Big Bang Theory*].

Otro nombre que tal vez le suene de la literatura (y de la televisión). En este caso, no hablamos de *Juego de Tronos*, sino del padre de la epidemiología moderna y líder en la introducción de medidas higiénicas en las prácticas médicas.

entrenamiento para clasificar nuevos datos. Por tanto, no requieren una fase de entrenamiento propiamente dicha.

Para construir clasificadores de este tipo, han de resolverse algunas cuestiones previas, entre las que destacan:

- ¿Cómo se realiza la clasificación?

Una forma sencilla de clasificar un caso es asignarle la misma clase que al caso más similar que conozcamos con una clase conocida. Es lo que hace el método de clasificación del vecino más cercano (1-NN). Si consideramos los k casos más similares, para evitar la excesiva influencia que podría tener un caso concreto mal etiquetado, entonces obtenemos un clasificador k -NN.

En este tipo de clasificadores, k es siempre impar en problemas de clasificación binaria (con dos clases). No tiene sentido probar con valores pares de k porque el error asociado a la regla k -NN es el mismo para $k = 2x$ y $k = 2x - 1$. De paso, evitamos posibles empates.

¿Qué valor elegimos para el parámetro k ? Dado que el clasificador escoge la clase más común entre los k vecinos más próximos, si escogemos un k demasiado pequeño, nuestro clasificador puede ser sensible a ruido en los datos. Si escogemos un k demasiado grande, entonces el vecindario de un punto puede incluir puntos de otras clases que podrían confundir a nuestro clasificador. Como todo en la vida, la virtud está en el término medio.

- ¿Qué casos deben utilizarse para clasificar?

Los clasificadores k -NN almacenan el conjunto de entrenamiento (o parte de él) y lo utilizan directamente para clasificar nuevos datos. Lo ideal sería almacenar aquellos casos típicos que recojan toda la información relevante que resulte necesaria para poder realizar una buena clasificación. Almacenar todos los casos conocidos (el conjunto de entrenamiento completo) podría hacer muy ineficiente el funcionamiento del clasificador, por lo que se puede mejorar su rendimiento con la ayuda de métodos de edición y condensado. Los métodos de edición (como la edición de Wilson o el Multiedit) intentan eliminar los patrones mal etiquetados que puedan aparecer cerca de las fronteras de decisión. Por su parte, los métodos de condensado (como el algoritmo de Hart) procuran reducir el número de muestras del conjunto de entrenamiento sin que esto afecte a la calidad del clasificador construido.

Un clasificador k -NN será más eficiente si empleamos métodos de condensado o, simplemente, utilizamos técnicas de indexación que optimicen la obtención de los vecinos más cercanos de un caso dado, como las propuestas por Fukunaga y Narendra.

- ¿Cómo se mide la similitud entre distintos casos?

Cuando los atributos son numéricos, se suele calcular la similitud entre casos utilizando alguna métrica de distancia (en el fondo, una medida de disimilitud), como la distancia euclídea o la distancia de Mahalanobis. Si utilizamos la distancia euclídea, sólo tenemos que calcular la raíz cuadrada de la suma de los cuadrados de las diferencias de los valores de los atributos. En ese caso, debemos usar factores de escala para asegurar que la influencia de todos los atributos sea similar (p.ej. normalizando los valores en el intervalo [0,1] o calculando z-scores).

Cuando los atributos son discretos, establecer una medida de similitud es bastante más problemático.

Si hay atributos irrelevantes, siempre se corre el riesgo de considerar muy diferentes casos que sólo difieren en los valores que toman los atributos irrelevantes para nuestro problema de clasificación. De ahí que pueda ser necesario emplear alguna técnica de selección de características.

Los clasificadores k -NN son muy sencillos y no requieren entrenamiento. Basta con escoger la clase más común entre los k vecinos más cercanos. Aunque puedan parecer extremadamente rudimentarios, Tom Cover y Peter Hart demostraron en 1967 que, con suficientes datos, el error cometido por un clasificador k -NN será, en el peor caso, sólo el doble del error cometido por el mejor clasificador imaginable, conocido como error de Bayes.⁶⁵ No está nada mal, un factor de aproximación 2 para un algoritmo tan sencillo.

El error cometido por un clasificador k -NN, obviamente, depende del valor de k . Cuanto mayor sea k , más se difuminarán las fronteras de decisión entre las distintas clases, hasta el punto de no ser demasiado útiles para clasificar (pensemos que, en el límite, cuando $k = n$, un clasificador k -NN se convierte en un clasificador por defecto que asigna a todo caso la clase más frecuente en los datos de entrenamiento). Puede resultar contraintuitivo pero, cuanto mayor sea el valor de k , menos complejo será nuestro modelo de clasificación. Cuanto mayor sea la variabilidad del clasificador resultante, mayor es su complejidad. Desde otro punto de vista: el modelo k -NN tiene más parámetros “ajustables” cuando $k = 1$.

En términos de sesgo y varianza, cuando k es pequeño, el error se debe a la varianza: las decisiones locales pueden cambiar drásticamente si lo hacen los datos que recopilemos. En el caso opuesto, si k es demasiado elevado, el error causado por no delimitar bien las fronteras de decisión se debe a que estamos introduciendo un sesgo excesivo en nuestro clasificador. Aumentar el valor del parámetro k , por tanto, aumenta el sesgo y disminuye la varianza, un caso típico del compromiso que hemos de

⁶⁵ Thomas M. Cover y Peter E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967. DOI: 10.1109/TIT.1967.1053964

alcanzar entre sesgo y varianza. En el caso del clasificador k -NN, incluso se puede derivar una expresión analítica de su error:

$$\begin{aligned} \text{error}(x) &= \text{bias}^2 + \text{varianza} + \text{error irreducible} \\ &= (f(x) - \frac{1}{k} \sum_{i=1}^k f(x_i))^2 + \frac{\sigma_\epsilon^2}{k} + \sigma_\epsilon^2 \end{aligned}$$

Se observa que la varianza es proporcional al error irreducible asociado al problema, σ_ϵ^2 , y decrece proporcionalmente con el número de vecinos considerado, k . El sesgo dependerá de lo suaves o abruptas que resulten las fronteras de decisión en nuestro problema: cuanto más abruptas sean, el sesgo del clasificador aumentará más cuando consideramos más vecinos en nuestra predicción.

Se han propuesto multitud de técnicas de aprendizaje automático que, en el fondo, no son más que diferentes reencarnaciones de los vecinos más cercanos:

- *Filtrado colaborativo*

En 1994, algunos investigadores de la Universidad de Minnesota y del MIT construyeron un sistema de recomendación basado en una idea engañosamente simple: las personas que mostraron gustos similares en el pasado es probable que los vuelvan a mostrar en el futuro.⁶⁶ Esa idea dio lugar a los sistemas de recomendación por filtrado colaborativo, omnipresentes hoy en día (p.ej. en Amazon⁶⁷).

- *Razonamiento basado en casos [CBR: Case-Based Reasoning]*

En determinadas aplicaciones, es importante establecer analogías para llegar a una solución. Pensemos, por ejemplo, en el sistema judicial anglosajón. A diferencia del derecho continental, la jurisprudencia es de vital importancia. Los jueces deben fundamentar sus decisiones haciendo un estudio minucioso de los precedentes, hechos o pruebas que incriminen al acusado. De ahí que los pasantes del despacho de abogados se pasen la noche en la biblioteca del juzgado buscando casos similares con sentencias judiciales favorables para sus intereses, algo que hemos visto en innumerables películas y novelas de John Grisham. Sistemas como Watson podrían encargarse perfectamente de realizar ese tipo de trabajos, aunque no quedaría tan bien en la ficción como sufrir con los abnegados abogados que luchan por una causa noble contra fuerzas más poderosas que ellos (a los que finalmente acaban derrotando).

- *LVQ [Linear Vector Quantization]*

Los métodos LVQ son métodos de aprendizaje adaptativo diseñados por Teuvo Kohonen, más conocido por sus mapas autoorganizativos. Se caracterizan por utilizar un número fijo y relativamente bajo de prototipos para aproximar las funciones de densidad de probabilidad

⁶⁶ Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, y John Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. En *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94, pages 175–186, 1994. ISBN 0-89791-689-1. DOI: 10.1145/192844.192905

⁶⁷ Greg Linden, Brent Smith, y Jeremy York. Amazon.Com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1): 76–80, January 2003. ISSN 1089-7801. DOI: 10.1109/MIC.2003.1167344

asociadas a las distintas clases de un problema de clasificación. Dada una secuencia de observaciones vectoriales (patrones), se selecciona un conjunto inicial de vectores de referencia (*codebooks* o prototipos). Iterativamente, se selecciona una observación x y se actualiza el conjunto de prototipos de forma que case mejor con x . Una vez finalizado el proceso de construcción del conjunto de prototipos (es decir, el entrenamiento del clasificador), las observaciones se clasificarán utilizando la regla 1-NN: buscando el vecino más cercano en el conjunto de vectores de referencia.

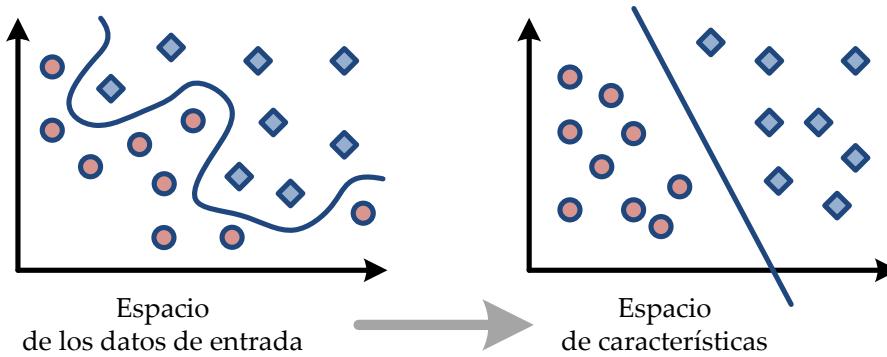
Algunos pueden ver los métodos LVQ como una versión muy particular de red neuronal pero, dadas sus semejanzas con los clasificadores 1-NN, aquí no los consideraremos como tales. Lo mismo se podría decir de la siguiente técnica de aprendizaje automático...

Máquinas de vectores de soporte (SVM)

Existen muchas técnicas estadísticas aplicables a problemas de clasificación, a las que los estadísticos se refieren con el término análisis discriminante. Estas técnicas suelen ser de tipo paramétrico: se asume la forma del modelo y, a partir de los datos de entrenamiento, se hallan los valores adecuados para los parámetros del modelo. Por ejemplo, un clasificador lineal asume que la clasificación puede realizarse mediante una combinación lineal de los valores de los atributos y emplea la combinación lineal que mejor se adapte al conjunto de casos de entrenamiento para clasificar nuevos casos. La frontera de decisión será una línea recta, un hiperplano si consideramos un espacio con d dimensiones, siendo d el número de atributos predictivos que se utilizan para clasificar. Un clasificador lineal utilizará, pues, una expresión de la forma $\alpha \cdot x = \sum_i \alpha_i x_i$ para diferenciar los ejemplos de una clase de los de otra.

En determinadas circunstancias, un clasificador cuadrático puede obtener mejores resultados que un clasificador lineal simple. Sin embargo, el ADC [Análisis Discriminante Cuadrático] requiere muchas más muestras de entrenamiento que el ADL [Análisis Discriminante Lineal] para obtener resultados similares. El ADC necesita estimar la matriz de covarianza de los datos, para lo que no siempre se dispone de suficientes muestras. En ocasiones, ni siquiera se puede aplicar porque no exista la inversa de esa matriz de covarianza, una condición necesaria en la construcción de un clasificador cuadrático.

Aunque, en teoría, el error de Bayes decrece conforme la dimensionalidad de los datos se incrementa, en la práctica disponemos de un conjunto fijo y finito de muestras para construir el clasificador (los estimadores están sesgados por las muestras disponibles). La bondad conseguida con un clasificador aumenta con la dimensionalidad de los datos hasta cierto punto, a partir del cual decrece conforme se incorporan nuevas



variables, algo conocido como el fenómeno de Hughes. Ese problema podría solucionarse consiguiendo más muestras de entrenamiento (lo cual no siempre es viable) o eligiendo un subespacio del espacio de patrones (mediante técnicas de selección de características).

Las máquinas de vectores de soporte, SVM, ofrecen una aproximación diferente al problema.^{68,69} Básicamente, se basan en construir un clasificador, lineal por ejemplo, pero sobre un espacio de características ampliado que no se corresponde con el espacio de características asociado a los atributos de los que disponemos en el conjunto de entrenamiento. De hecho, son incluso capaces de trabajar con un conjunto infinito de características.

Supongamos que queremos construir un clasificador lineal. Cada uno de los ejemplos de nuestro conjunto de entrenamiento lo podemos ver como un punto en un espacio d -dimensional. La frontera de decisión será un hiperplano de $d - 1$ dimensiones que nos permita separar los ejemplos de una clase de los de otra. Ahora bien, de todos los posibles hiperplanos que pueden separar las clases, ¿con cuál nos quedamos? Parece lógico que seleccionemos aquél que corresponda a una separación máxima entre las clases. Esa separación, denominada margen, será máxima cuando se maximice la distancia de la frontera de decisión a los ejemplos más próximos de cada clase. Esos ejemplos serán nuestros vectores de soporte.

El problema es que las diferentes clases de un problema raramente serán linealmente separables. De ahí que ampliemos nuestro conjunto de atributos con características adicionales que hagan nuestro problema de clasificación más sencillo en un espacio de dimensionalidad mayor en el que las clases sí sean linealmente separables.

El truco de los clasificadores SVM está en que, aunque pueden trabajar con una familia infinita de características, no resulta necesario que las construyan de forma explícita. Las máquinas SVM se diseñan para trabajar directamente con productos escalares de los vectores de características de los ejemplos del conjunto de entrenamiento.⁷⁰ Mediante la formulación de un problema dual de optimización que es matemáticamente equivalente a maximizar el margen sobre el espacio de características

Figura 18: Máquina de vectores de soporte, SVM [*Support Vector Machine*]. Del espacio de características de los datos de entrenamiento se deriva un espacio de características, de mayor dimensionalidad, en el que resulta más sencillo establecer una frontera de decisión entre las distintas clases.

⁶⁸ Bernhard Scholkopf y Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001. ISBN 0262194759

⁶⁹ Nello Cristianini y John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000. ISBN 0521780195

⁷⁰ William H. Press, Saul A. Teukolsky, William T. Vetterling, y Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007. ISBN 0521880688

ampliado.

Las máquinas SVM emplean el denominado truco del kernel [*kernel trick*]. Utilizan funciones kernel $k(x, y)$, que no son más que unas funciones de similitud definidas sobre los conjuntos de características del conjunto de entrenamiento (de ahí que los hayamos incluido en la familia de técnicas analógicas). Con ayuda de esas funciones, los puntos x_i del conjunto de entrenamiento se comparan con los puntos del hiperplano que hace de frontera de decisión en el espacio de características ampliado. Una vez entrenado el clasificador para obtener el hiperplano de máximo margen, ese hiperplano vendrá caracterizado por una expresión de la forma $\sum_i \alpha_i k(x_i, x)$, donde los x_i representan a los ejemplos del conjunto de entrenamiento. Dado que $k(x_i, x)$ es una medida de similitud, cada término de la suma mide cómo de cerca está el punto x del ejemplo de entrenamiento x_i y la combinación lineal de esos términos mide la cercanía relativa del ejemplo que queremos clasificar x a los ejemplos que quedan a ambos lados de la frontera de decisión, que puede adoptar una forma muy enrevesada en el espacio de características original, lo que permite a la máquina SVM resolver problemas de clasificación en los que las clases no son linealmente separables en su espacio original.

Por el lado positivo, las máquinas SVM ofrecen una precisión generalmente alta en comparación con otros métodos de clasificación. Además, suelen ser bastante robustas frente a la presencia de ruido en los datos. Por el lado negativo, las máquinas SVM son más costosas de entrenar que otros métodos. Su eficiencia y escalabilidad es menor que, por ejemplo, la asociada a los clasificadores *k*-NN. Además, resultan difíciles de interpretar, al estar basadas en la realización de transformaciones matemáticas para conseguir que las clases sean linealmente separables, transformaciones a las que difícilmente podremos dar una interpretación semántica en términos de los atributos presentes en nuestro conjunto de entrenamiento.

Las máquinas SVM se hicieron muy populares a finales del siglo XX y, tal vez, hayan sido el método de clasificación más utilizado durante la primera década del siglo XXI. Sucedieron a los árboles de decisión como técnica más popular y ahora están siendo reemplazadas por las técnicas de *deep learning* como método favorito para la resolución de problemas de aprendizaje automático.

Técnicas bayesianas: Modelos probabilísticos

Llegamos a la tercera gran familia de técnicas de aprendizaje automático: los modelos probabilísticos.⁷¹ El conocido teorema de Bayes es la piedra angular de las técnicas probabilísticas, de ahí que los que trabajan

⁷¹ Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012. ISBN 0262018020

en este tipo de técnicas reciben el apelativo de bayesianos:

$$p(y|x)p(x) = p(x|y)p(y)$$

Reordenando los términos, obtenemos la forma en la que resulta más útil en aprendizaje automático:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

Con esta expresión, podemos predecir la probabilidad $p(y|x)$ de una clase y dado un caso x disponiendo únicamente de información asociada a la probabilidad de los casos $p(x)$, la probabilidad de las clases $p(y)$ y la probabilidad $p(x|y)$ de los casos dada la clase. Todas esas probabilidades las podemos estimar con facilidad a partir de los datos de nuestro conjunto de entrenamiento.

En términos más generales, el teorema de Bayes nos permite realizar inferencias de tipo probabilístico. Mediante la inferencia bayesiana, podemos estimar la probabilidad $p(y|x)$ de la hipótesis y dada la evidencia x a partir de una función $p(x|y)$ de verosimilitud [*likelihood*] que nos indica la probabilidad de que se cumpla la evidencia x si la hipótesis y es cierta. Esa función de verosimilitud, que se puede derivar de un modelo estadístico de los datos observados, nos proporciona un mecanismo mediante el cual actualizar la probabilidad de la hipótesis y dada la evidencia x . El proceso de inferencia nos permite pasar de la probabilidad a priori $p(y)$ a la probabilidad a posteriori $p(y|x)$, incorporando la información proporcionada por la presencia de la evidencia x , cuya probabilidad marginal $p(x)$ es necesario conocer para completar la fórmula del teorema de Bayes.

Para realizar inferencias, los bayesianos comienzan con una creencia previa [*prior*] acerca de la probabilidad a priori $p(y)$. Cuando se obtienen nuevos datos, se actualiza esa creencia incorporando los datos observados [*posterior*] al calcular la probabilidad a posteriori $p(y|x)$. De forma iterativa, esa creencia se sigue actualizando conforme se procesan más datos. Esa es la base, por ejemplo, de los filtros anti-spam que nos permiten consultar nuestro correo electrónico sin que nuestro buzón se vea saturado de correo no deseado.

Entre los bayesianos más relevantes se encuentra Judea Pearl, premio Turing de la Universidad de California en Los Ángeles (UCLA). Él fue el que propuso las redes bayesianas en 1985 para enfatizar la naturaleza subjetiva de la información y el uso del teorema de Bayes como mecanismo para actualizar esa información. Michael I. Jordan, de la Universidad de California en Berkeley, y David Heckerman, de Microsoft Research, son otros de los investigadores con más renombre del área.

Las técnicas probabilísticas se pueden utilizar tanto en aprendizaje supervisado como en aprendizaje no supervisado:

El teorema de Bayes recibe su nombre del reverendo presbiteriano inglés Thomas Bayes, que en realidad nunca publicó en vida el trabajo por el que pasaría a la historia. Fue el predicador unitario galés Richard Price el que recopiló y editó los manuscritos inéditos de Bayes, publicados de forma póstuma en 1763, un par de años después de la muerte de Bayes. Price defendía que el teorema de Bayes ayudaba a demostrar la existencia de Dios usando un argumento teleológico: la existencia de Dios (deidad o creador inteligente) a partir de la evidencia percibida en forma de orden y regularidades en el mundo natural; esto es, Dios como causa última de un diseño deliberado. De forma aparentemente independiente, el matemático francés Pierre-Simon, marqués de Laplace, derivó los mismos resultados en 1774.

Michael Jordan, el informático, fue uno de los editores de la revista *Machine Learning* que dimitieron en 2001 por estar en contra del modelo de negocio de las revistas académicas de pago y se pasaron al *Journal of Machine Learning Research* (JMLR), revista científica abierta en la que los autores retienen el copyright de sus trabajos (sin pagar nada) y cualquiera puede acceder a sus archivos en Internet: <http://www.jmlr.org/>

- En aprendizaje no supervisado, p.ej. clustering, tenemos algo parecido al dilema del huevo y la gallina: “¿qué fue primero, el huevo o la gallina?”. Si supiésemos las clases a las que pertenecen los casos (sus agrupamientos), podríamos modelar las clases a partir de los ejemplos. Si conociésemos los modelos de las clases, podríamos inferir las clases de los ejemplos (sería un problema de aprendizaje supervisado). Pero desconocemos tanto lo primero como lo segundo.

Sin embargo, podemos salir del atolladero de la siguiente forma. En primer lugar, adivinamos la clase a la que pertenece cada ejemplo, aunque lo hagamos de forma aleatoria. A partir de los datos y las clases que les hemos asignado, podemos aprender un modelo de esas clases. Una vez que tenemos un modelo de las clases, podemos reasignar los casos a las clases con las que mejor encajen. Y vuelta a empezar... El proceso se repite iterativamente hasta que se estabilice la asignación de ejemplos a clases. ¿Le suena de algo? Es la misma idea que emplea el algoritmo de las k medias o k -means.

¿Funciona de verdad? Tres estadísticos de Harvard, Arthur Dempster, Nan Laird y Donald Rubin, lo demostraron en 1977.⁷² En cada iteración de su algoritmo de maximización de expectativas [*EM: Expectation Maximization*], el modelo de los agrupamientos es mejor y el proceso termina cuando se alcanza un máximo local de la función de verosimilitud. El algoritmo se denomina EM porque en cada iteración se calculan esperanzas (término estadístico para hacer referencia al valor esperado de la función de verosimilitud, i.e. la descripción de las clases) y se maximiza (calculando los estimadores de máxima verosimilitud de los parámetros de nuestro modelo, i.e. asignando clases a los ejemplos).

- En aprendizaje supervisado, el teorema de Bayes nos indica cómo podemos calcular la probabilidad de que un ejemplo pertenezca a una clase concreta:

$$p(y = y_j|x) \propto p(x|y = y_j)p(y_j)$$

La expresión anterior, que se deriva directamente del teorema de Bayes suprimiendo la probabilidad marginal $p(x)$, es todo lo que tenemos que evaluar si sólo nos interesa determinar cuál es la clase más probable para nuestro ejemplo x .

El único problema es que, tal como lo hemos definido, nuestro clasificador bayesiano necesitaría un número exponencial de parámetros. Supongamos que nuestros ejemplos x vienen descritos por d atributos, cada uno de los cuales puede tomar v valores diferentes ($v = 2$ para atributos binarios). Para calcular $p(x|y = y_j) = p(x_1..x_d|y = y_j)$, necesitaremos estimar $v^d - 1$ parámetros. Uno nos lo ahorraremos porque sabemos que las probabilidades deben sumar 1 en total. Aun así, si

⁷² Arthur P. Dempster, Nan M. Laird, y Donald B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society (Series B: Methodological)*, 39(1):1–38, 1977. URL <http://www.jstor.org/stable/2984875>

nuestro problema de clasificación tiene k clases diferentes, en total tendremos que estimar $(k - 1)(v^d - 1)$ parámetros, con $v \geq 2$, sólo para las verosimilitudes $p(x|y = y_j)$. Por muy grande que sea nuestro conjunto de entrenamiento, el número de parámetros de nuestro modelo podría ser mucho mayor, lo que imposibilitaría conseguir una estimación fiable de los valores del número exponencial de parámetros que formarían parte de nuestro modelo probabilístico.

Naïve Bayes

Una forma sencilla de solucionar el problema de estimar la probabilidad conjunta $p(x_1..x_d|y = y_j)$, lo que requiere el ajuste de un número exponencial de parámetros, es asumir que las variables mediante las que se describen nuestros ejemplos son independientes.

Cuando dos variables A y B son independientes, entonces la probabilidad de A condicionada por B es igual a la probabilidad de A : $p(A|B) = p(A)$. Por tanto, por la propia definición de la probabilidad condicionada, $p(A|B) = p(AB)/p(B)$, obtenemos que $p(AB) = p(A)p(B)$.

Si añadimos a nuestro modelo de clasificación probabilístico la suposición de que todas las variables son independientes, entonces obtenemos el conocido clasificador *Naïve Bayes* (el Bayes “ingenuo”):

$$p(y = y_j|x) \propto p(x|y = y_j)p(y_j)$$

Como cada ejemplo x viene descrito por un conjunto de atributos $(x_1..x_d)$ que asumimos que son independientes:

$$p(x|y = y_j) = p(x_1|y = y_j) \times \dots \times p(x_d|y = y_j) = \prod_{i=1}^n p(x_i|y = y_j)$$

Este modelo de clasificación nos permite que analicemos por separado cada atributo. Para estimar $p(x_i|y = y_j)$ sólo tendremos que ajustar $v - 1$ parámetros, $(k - 1)(v - 1)$ en un problema de clasificación con k clases. Como nuestros ejemplos vienen descritos por d atributos que asumimos que son independientes, nos bastará un total de $(k - 1)d(v - 1)$ parámetros para modelar la función de verosimilitud $p(x|y = y_j)$. Un número lineal de parámetros con respecto al número de valores distintos que aparecen en el conjunto de entrenamiento (frente al número exponencial necesario para modelar la distribución de probabilidad conjunta sin la suposición de independencia).

La principal ventaja del clasificador Naïve Bayes es que podemos entrenarlo (ajustar sus parámetros) con un simple recorrido secuencial de nuestro conjunto de entrenamiento, ya que las probabilidades $p(x_i|y = y_j)$ se pueden estimar directamente a partir de la frecuencia de aparición de cada valor del atributo x_i para cada clase y_j .

Sólo hemos de tener cuidado con situaciones en las que un valor concreto de x_i no aparece para ningún ejemplo de la clase y_j . Para evitar que nuestra estimación de la probabilidad $p(x_i|y = y_j)$ sea cero, se suele emplear algún tipo de suavizado en la estimación. Por ejemplo, el suavizado de Laplace nos permite estimar esa probabilidad como

$$p(x_i|y = y_j) = \frac{n(x_i y_j) + 1}{\sum_v (n(x_v y_j) + 1)} = \frac{n(x_i y_j) + 1}{n(y_j) + v}$$

suponiendo que v es el número de valores distintos del atributo x_i y ausencia de valores nulos para x_i ; esto es, que el atributo x_i siempre tiene algún valor para todos los ejemplos del conjunto de entrenamiento correspondientes a la clase y_j . Si el atributo presenta valores nulos, entonces $\sum_v n(x_v y_j) < n(y_j)$ y no podríamos simplificar la expresión anterior eliminando la sumatoria.

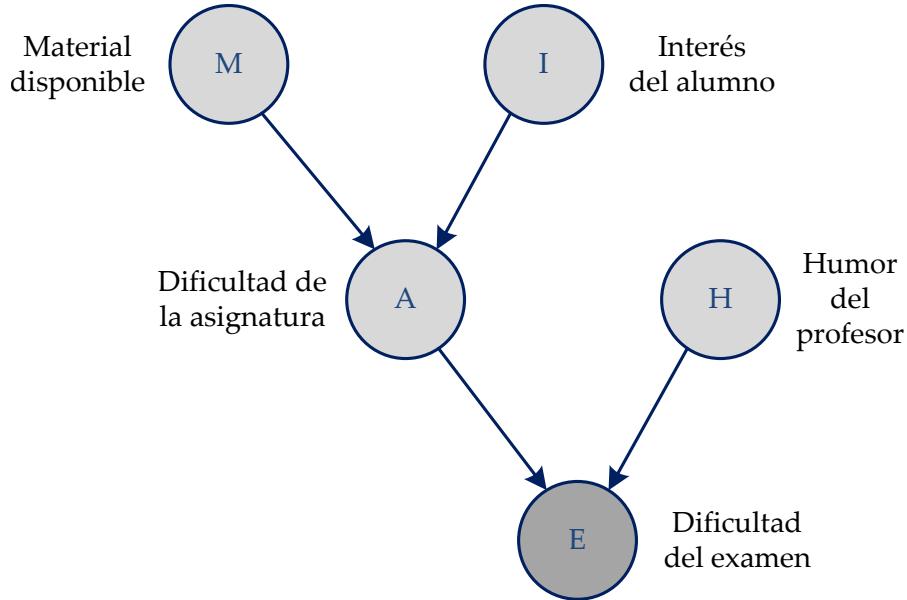
La eficiencia del entrenamiento de los clasificadores Naïve Bayes los hace muy atractivos para problemas en los que el entrenamiento de otros modelos de clasificación podría resultar excesivamente costoso, ya sea por la dimensionalidad de los datos, por el tamaño del conjunto de datos de entrenamiento o porque el clasificador tenga que enfrentarse a un contexto cambiante. No resulta extraño encontrar clasificadores de este tipo en problemas de clasificación con miles de características, como puede ser la clasificación de documentos en los que la presencia o ausencia de un término concreto puede usarse como característica del documento a la hora de clasificar. De ahí que el humilde Naïve Bayes pueda que se encuentre detrás del filtro anti-spam que le permite usar su correo electrónico sin demasiadas intromisiones no deseadas.

En términos del error cometido, reducir el número de parámetros de un clasificador probabilístico, como hace Naïve Bayes, permite reducir su varianza a costa de incrementar su sesgo. Al menos, en principio. En la práctica, la reducción de varianza de un clasificador sencillo como Naïve Bayes cancela parte del sesgo introducido por el modelo. Este hecho explicaría por qué un método tan simple es, en ocasiones, competitivo con otros métodos de clasificación mucho más sofisticados. A veces, incluso, superior a ellos.

En contrapartida a su eficiencia, el clasificador Naïve Bayes realiza una suposición que puede no ser válida. Presupone que todas las variables con las que vienen descritos los ejemplos son independientes cuando, de hecho, lo más normal es que no lo sean. Las redes bayesianas nos pueden ayudar en tales situaciones.

Redes bayesianas

Las redes bayesianas son modelos probabilísticos que generalizan el método Naïve Bayes. Para evitar tener que construir un modelo probabilístico con un número exponencial de parámetros, las redes bayesianas



nos permiten especificar qué variables son independientes y cuáles no lo son.

Las dependencias entre variables se representan gráficamente mediante un grafo dirigido acíclico. En una red bayesiana, los nodos representan las variables de nuestro problema y los arcos entre los nodos representan las dependencias que existen entre parejas de variables. Una variable dada x dependerá de sus padres en la red bayesiana, por lo que tendremos que estimar los parámetros de una distribución de probabilidad condicionada $p(x|padres(x))$. En el ejemplo de la figura, la dificultad de la asignatura depende de la calidad del material disponible y del interés que una asignatura despierta en el alumno, por lo que en el nodo asociado a la asignatura tendremos que modelar la distribución de probabilidad condicionada $p(A|M, I)$. De forma similar, la dificultad del examen se modelará con una distribución $p(E|A, H)$. Por último, las variables correspondientes a nodos sin padres en la red se modelan directamente como $p(M)$, $p(I)$ y $p(H)$.

De esta forma, una red bayesiana ofrece una flexibilidad mucho mayor que un clasificador Naïve Bayes, a la vez que las independencias identificadas entre las diferentes variables ayudan a reducir el número de parámetros necesario para describir la distribución de probabilidad conjunta de todas las variables del problema. Las redes bayesianas son universales en el sentido de que, si se dispone de suficientes datos, son capaces de aproximar cualquier distribución de probabilidad.

¿Cómo se entrena una red bayesiana?

- Dada la estructura de la red, sólo tenemos que estimar los valores de los parámetros que describen las probabilidades condicionadas

Figura 19: Red bayesiana para predecir la dificultad de un examen, la cual depende de la dificultad de la asignatura y del estado de humor del profesor el día que preparó el enunciado. A su vez, la dificultad de una asignatura depende del material disponible y del interés del alumno: cuanto mejor sea el material disponible, más fácil resulta prepararse la asignatura; cuanto más interés tenga el alumno, más sencilla le parecerá la asignatura. ¿Cuántos parámetros se necesitan para modelar esta red bayesiana? ¿Cuántos harían falta para modelar la distribución de probabilidad conjunta $p(E, A, H, M, I)$? Ahí reside la ventaja que ofrecen las redes bayesianas.

asociadas a cada nodo de la red (tan sencillo como el entrenamiento de Naïve Bayes).

- Dada la estructura de la red, cuando ésta incluye algunas variables ocultas (no observadas), se busca una configuración adecuada de la red que encaje con los datos observados en nuestro conjunto de entrenamiento. Para ello, se suele recurrir a técnicas de optimización como el gradiente descendente (algo más complejo que el entrenamiento de Naïve Bayes).
- Dadas únicamente las variables observables (las que aparecen en el conjunto de entrenamiento), se intenta determinar cuál debería ser la topología de la red bayesiana (algo que puede ser extremadamente ineficiente y costoso en términos computacionales).

Las redes bayesianas se han empleado con relativo éxito en la resolución de algunos problemas prácticos reales, como en el asistente para la resolución de problemas de Windows (en ocasiones, puede resultar útil para detectar e identificar la causa del problema) o los agentes incluidos en algunas versiones antiguas de Office (el famoso clip animado, que estorbaba más que ayudaba).

Redes de Markov

Durante las últimas dos décadas, un modelo probabilístico diferente ha ido reemplazando a las redes bayesianas en aprendizaje automático: las redes de Markov, también conocidas como “campos aleatorios de Markov” [MRF: *Markov Random Fields*].

A diferencia de las redes bayesianas, que forman grafos dirigidos acíclicos, las redes de Markov son modelos gráficos no dirigidos. Una red de Markov viene descrita por un grafo no dirigido en el que los nodos siguen representando variables y las aristas tienen pesos asociados. Conjuntamente, variables y pesos describen una distribución de probabilidad.

Las redes bayesianas, al obligarnos a elegir la dirección de las dependencias, pueden ser útiles para representar relaciones causales. De hecho, cuando lo hacen se les suele denominar redes causales. Sin embargo, existen situaciones en las que un modelo no dirigido puede resultar más natural. Por ejemplo, si queremos representar que el brillo de un píxel en una imagen está correlacionado con la intensidad de los píxeles adyacentes, una red bayesiana introduciría dependencias poco naturales. Un modelo gráfico no dirigido, como una red de Markov, no requiere asociarle una dirección a las conexiones entre variables, por lo que resulta mucho más natural para modelar este tipo de situaciones.

¿Qué ventajas tienen las redes de Markov frente a las bayesianas?

- Son modelos simétricos y, por tanto, más “naturales” para determinados dominios (p.ej. a la hora de modelar relaciones espaciales).

- Las redes de Markov discriminativas, conocidas como “campos aleatorios condicionales” o CRF [*CRF: Conditional Random Fields*], permiten definir distribuciones de probabilidad condicional $p(y|x)$ y suelen funcionar mejor que sus equivalentes bayesianas.

¿Qué desventajas tienen las redes de Markov frente a las bayesianas?

- Sus parámetros son menos interpretables y, las redes resultantes, menos modulares que las bayesianas.
- La estimación de sus parámetros es computacionalmente más costosa que la de los parámetros de las redes bayesianas.

Técnicas evolutivas: Computación evolutiva

Llegamos a la cuarta de las familias de técnicas de aprendizaje automático, la formada por los “evolutivos”.

La computación evolutiva [*EC: Evolutionary Computation*]⁷³ como rápidamente habrá podido inferir, se inspira en la teoría de la evolución de Charles Darwin. Darwin publicó su teoría en su libro “Sobre el origen de las especies por medio de la selección natural”, que escribió años después de su famoso viaje por las islas Galápagos en el Beagle. Según Darwin, los individuos son ligeramente distintos entre sí y estas pequeñas variaciones hacen que cada uno tenga distintas capacidades para adaptarse a su medio ambiente, así como para reproducirse y transmitir sus rasgos a sus descendientes. Con el paso del tiempo, tras varias generaciones, los rasgos de los individuos que mejor se adaptaron a las condiciones de su entorno se vuelven más comunes, haciendo que la población, en su conjunto, evolucione. Del mismo modo, la naturaleza va seleccionando las especies mejor adaptadas para sobrevivir y reproducirse: la “selección natural”. Pequeños cambios heredables y la lucha por la supervivencia son los factores que provocan cambios en la Naturaleza y la aparición de nuevas especies.

Las técnicas evolutivas se basan en el uso de una función de aptitud o *fitness* que ayuda a guiar el proceso de búsqueda de soluciones para un problema. A diferencia de otras técnicas de búsqueda, suelen utilizar poblaciones de soluciones potenciales en vez de un único individuo, lo cual las hace menos sensibles a quedar atrapadas en óptimos locales. Como técnicas heurísticas que son, no obstante, los algoritmos evolutivos no garantizan la optimalidad de las soluciones encontradas. En muchas ocasiones, ni siquiera se puede determinar cómo de cerca del óptimo se encuentra la solución particular obtenida por medio de técnicas evolutivas.

Otra diferencia con respecto a técnicas tradicionales de búsqueda es que las técnicas evolutivas usan operadores probabilísticos, mientras que

⁷³ Agoston E. Eiben y James E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2nd edition, 2015. ISBN 3662448734

las tradicionales utilizan operadores determinísticos. Aunque las técnicas evolutivas sean estocásticas, el hecho de que usen operadores probabilísticos no significa que operen de manera análoga a una simple búsqueda aleatoria. Muy criticadas por investigadores de la escuela “simbólica”, se creía que una simple búsqueda aleatoria podía superarlas. De hecho, fue un incidente en un congreso lo que ocasionó la segregación de los “evolutivos”. En el congreso internacional de aprendizaje automático de 1995 [*ICML: International Conference on Machine Learning*], en el lago Tahoe, California, Kevin Lang publicó un artículo que mostraba que la ascensión de colinas vencía a las técnicas evolutivas en los mismos problemas. Previamente, algunos investigadores “evolutivos” habían intentado publicar sus resultados en el ICML, un congreso de reconocido prestigio, pero continuamente les rechazaban sus artículos por considerar insuficiente su validación empírica. Al asistir a la presentación de Lang, John Koza se indignó y redactó rápidamente un artículo de 23 páginas a dos columnas en el formato estándar del ICML, del que dejó una copia en cada asiento del auditorio en el que se celebraba el congreso. Koza refutaba las conclusiones de Lang y acusaba a los revisores del ICML de mala conducta científica. El incidente del lago Tahoe marcó el divorcio entre los “evolutivos” y el resto de los investigadores en aprendizaje automático. Koza y los suyos crearon su propio congreso, que se fusionó con otro dedicado a los algoritmos genéticos, ICGA [*International Conference on Genetic Algorithms*], para dar lugar a GECCO [*Genetic and Evolutionary Computing Conference*], el congreso más importante del área desde 1999. Algo que no contribuyó demasiado a que el resto de los especialistas en aprendizaje automático dejaran de considerar “mal fundamentadas” e “inestables” a las técnicas evolutivas.

Las distintas técnicas evolutivas son metaheurísticas, por lo que llevan implícito el método para resolver un problema. En general, los algoritmos evolutivos son muy simples, fáciles de implementar y paralelizables. Son independientes del problema, lo que las hace robustas, por ser útiles para cualquier problema, pero a la vez débiles, pues no están especializadas en ninguno en particular.

Las técnicas evolutivas tienen la ventaja de que no necesitan disponer de conocimiento específico sobre el problema que intentan resolver. No obstante, disponer de él puede ayudar. Ofrecen la posibilidad de incorporar conocimiento sobre el dominio del problema que se pretende resolver mediante su hibridación con otras técnicas de optimización, como sucede con los algoritmos lamarckianos y baldwinianos.

■ *Lamarckismo*

Jean-Baptiste Pierre Antoine de Monet, Chevalier de Lamarck, más conocido por Jean Baptiste Lamarck (1744-1829) fue un naturalista francés que desarrolló la primera teoría evolutiva coherente cincuenta

años antes que Darwin, en 1809. Según Lamarck, los organismos pueden adquirir nuevas características por la influencia de su entorno y los caracteres adquiridos se pueden heredar. Esto es, los organismos tendrían la capacidad de trasladar a sus descendientes los caracteres adquiridos en vida. Hoy conocemos mucho más acerca de la biología que en época de Lamarck y sabemos que el Lamarckismo no se sostiene. De la misma forma que el hijo de un pianista no nace sabiendo tocar el piano, una jirafa no estira su cuello para llegar a su comida y eso hace que sus descendientes hereden su cuello ya estirado. De ahí que los caricaturistas satíricos dibujasen a Lamarck con un cuello de jirafa (de la misma forma que dibujaban a Charles Darwin como un mono).

El monje agustino Gregor Mendel aún no había nacido cuando Lamarck publicó su teoría. Los experimentos con guisantes de Mendel le permitieron descubrir que los caracteres se heredaban de forma discreta, del padre o de la madre, dependiendo de su carácter dominante o recesivo. Estos caracteres recibieron el nombre de genes, que podían tomar diferentes valores. Los valores que podían tomar, alelos. Ya en el siglo XX, se determinó que los genes están en el ADN de los cromosomas del núcleo de la célula (y en el ADN mitocondrial, que se hereda por vía materna). La evolución natural hace que se vaya alterando la proporción de alelos de un tipo determinado en una población. Algunos factores disminuyen su variabilidad, como la elección natural y la deriva genética, mientras que otros la aumentan, como sucede con la mutación, la poliploidía, la recombinación o cruce y el flujo genético.

Sin embargo, aunque biológicamente se haya descartado, la teoría de Lamarck se puede emplear en un algoritmo evolutivo de optimización. En cada generación, podemos usar técnicas de optimización local para mejorar los individuos de una población y que sean esos individuos mejorados los que intervengan en la formación de la siguiente generación de individuos.

■ *El efecto Baldwin*

Según James Mark Baldwin, psicólogo y filósofo norteamericano, las condiciones genéticas transmisibles por herencia pueden hacer más fácil el aprendizaje de técnicas y trucos que sólo poseen aquellos que tengan una variante evolutiva determinada. La evolución báldwiniana, propuesta en 1896, indica que el comportamiento sostenido de una especie o grupo puede guiar la evolución de esa especie: Las habilidades que inicialmente requieren el aprendizaje son finalmente reemplazadas por la evolución de sistemas genéticamente determinados que no requieren aprendizaje. Primero se aprenden comportamientos y luego se fijan genéticamente. Los lobos que aprendieron a ser más sociables se empezaron a acercar al hombre y, con el tiempo, se convirtieron en

perros.

El efecto Baldwin, en suma, reconoce que el aprendizaje individual puede explicar fenómenos evolutivos que parecen apoyar la herencia lamarckiana, pero lo hace sin recurrir a ella. En la evolución baldwiniana, también conocida como evolución ontogenética, las condiciones epigenéticas pueden ser igual o más importantes que la selección natural. La plasticidad fenotípica de un organismo hace referencia a su capacidad para adaptarse a su ambiente durante su tiempo de vida; esto es, su capacidad de aprendizaje. La descendencia seleccionada tenderá hacia tener una mayor capacidad de aprender nuevas habilidades, sin estar limitada a habilidades genéticamente codificadas y relativamente fijas. Eso explicaría por qué el hombre, sin ser el más fuerte ni el más rápido, ha conseguido llegar a la cima de la evolución (al menos, por el momento).

Volviendo al tema del aprendizaje automático. Un algoritmo evolutivo de tipo baldwiniano utilizaría técnicas de optimización local para mejorar el *fitness* de los individuos de una población, pero esa mejora no se heredaría directamente. El efecto Baldwin explicaría por qué se tenderá a seleccionar aquellos individuos que más capaces sean luego de mejorar por sí mismos; esto es, de aprender. Geoffrey Hinton y Steven Nowlan demostraron el efecto Baldwin mediante un algoritmo genético diseñado para hacer evolucionar la estructura de una red neuronal.⁷⁴ Observaron que el *fitness* aumentaba con el tiempo sólo si se permitía que los individuos pudiesen aprender.

Dentro de la familia de los “evolutivos”, los investigadores más conocidos son John Holland, de la Universidad de Michigan; Lawrence Fogel, que trabajó para General Dynamics; Ingo Rechenberg, de la Universidad Técnica de Berlín; y John Koza, que trabajó en Stanford antes de fundar Genetic Programming Inc. y Scientific Games Corporation. Sus trabajos dieron lugar, respectivamente, a las cuatro técnicas evolutivas más conocidas: los algoritmos genéticos, la programación evolutiva, las estrategias de evolución y la programación genética. Aparte de ellas, existen otras técnicas que pueden considerarse evolutivas, al estar basadas en el comportamiento de los seres vivos en la Naturaleza. Es el caso de la inteligencia de enjambres [*swarm intelligence*], los sistemas inmunes artificiales [*artificial immune systems*] o las técnicas de optimización basadas en colonias de hormigas [*ant colony optimization*].

Algoritmos genéticos

John Henry Holland, que en 1959 consiguió el primer título de Doctor en Informática concedido en el mundo, pretendía conseguir que los ordenadores aprendiesen imitando el proceso de la evolución. Aunque

⁷⁴ Geoffrey E. Hinton y Steven J. Nowlan. How learning can guide evolution. *Complex Systems*, 1(3): 495–502, 1987. ISSN 0891-2513. URL http://www.complex-systems.com/abstracts/v01_i03_a06.html

el biólogo inglés Alex S. Fraser realizó simulaciones de la evolución de sistemas biológicos en un ordenador digital y el biofísico alemán Hans-Joachin Bremermann interpretó la evolución como un proceso de optimización en el que cadenas de bits se reproducían, se seleccionaban y mutaban, fue John Holland el primero que utilizó la teoría de la evolución para diseñar algoritmos de aprendizaje automático.

Tras leer un libro del biólogo evolutivo Ronald Aylmer Fisher, "La teoría genética de la selección natural", Holland decidió imitar los procesos adaptativos de los sistemas naturales y diseñar sistemas artificiales (programas) que retengan los mecanismos de los sistemas naturales.

En un algoritmo genético, se hace evolucionar una población de individuos, cada uno de los cuales representa una posible solución. Esa población se somete a acciones aleatorias semejantes a las de la evolución biológica (mutaciones y recombinaciones genéticas). De una generación a otra, los individuos se seleccionan de acuerdo con una función de adaptación o *fitness*, en función de la cual se decide qué individuos sobreviven (los más adaptados) y cuáles son descartados (los menos aptos).

La selección se realiza siempre de forma probabilística. Un individuo es más probable que se seleccione si tiene un mejor valor de *fitness*, aunque cualquier individuo, por malo que sea, tiene una probabilidad de selección estrictamente mayor que cero. En ocasiones, si no queremos que una muy buena solución encontrada en una generación intermedia de la evolución se pierda por el camino, podemos introducir cierto grado de elitismo en la selección. Por ejemplo, podemos hacer que el mejor individuo de la población (o los *k* mejores) siempre sobrevivan, algo que aleja a los algoritmos genéticos del mundo real, donde una generación termina siempre reemplazando por completo a las anteriores.

El secreto está en conseguir el equilibrio adecuado entre exploración y explotación. Exploración del espacio de posibles soluciones mediante mutaciones aleatorias y explotación de las cualidades de las mejores soluciones obtenidas hasta el momento mediante la recombinación de los mejores individuos de la población, seleccionados estocásticamente para dar forma a la siguiente generación.

Según la hipótesis de la Reina Roja (en referencia a la obra de Lewis Carroll, "A través del espejo"), la adaptación de los individuos es necesaria en la naturaleza, no sólo para obtener una ventaja reproductiva, sino para sobrevivir con otros organismos que también evolucionan en un entorno siempre cambiante. La reproducción sexual sería, pues, necesaria para mantener cierta variedad en la población de individuos en su lucha contra parásitos, de forma que ningún germe pueda infectar a todos sus individuos. En ausencia de la presencia de otros organismos, por tanto, la reproducción sexual sería irrelevante en el aprendizaje automático. Danny Hillis, diseñador del supercomputador paralelo Connection Machine en el MIT y cofundador de Thinking Machines Corporation, la

La evolución también se ha empleado para explicar la creatividad humana, como hizo el psicólogo William James ("Great Men, Great Thoughts, and the Environment", The Atlantic Monthly, 1880): la innovación como resultado de la generación aleatoria de ideas y de mentes astutas capaces de identificar y retener las mejores. El también psicólogo Donald Campbell atribuye todos los avances en el conocimiento humano a un proceso evolutivo de variación ciega y retención selectiva ("Blind Variation and Selective Retention in Creative Thought as in Other Knowledge Processes", Psychological Review, 1960).

Según Pedro Domingos, la reproducción sexual dejaría de ser necesaria, al menos, mientras los algoritmos de aprendizaje no tengan que competir con virus informáticos por el tiempo de CPU y el espacio en memoria.

empresa que lo comercializó, sostiene que introducir deliberadamente parásitos que coevolucionen en un algoritmo genético puede ayudar a escapar de óptimos locales incrementando gradualmente la dificultad del entorno en el que los individuos han de prosperar. Según Christos Papadimitriou, informático teórico de Berkeley, la reproducción sexual no ayuda a optimizar el *fitness* de los individuos, sino su ‘mezclabilidad’: la capacidad de un gen de funcionar correctamente, en media, cuando se combina con otros muchos genes.

La aplicación de un algoritmo genético consiste en hallar de qué parámetros depende el problema, codificarlos adecuadamente en un cromosoma y aplicar los métodos de la evolución: selección y reproducción sexual con intercambio de información y alteraciones que generen diversidad. La mayoría de las veces, una codificación acertada es la clave para que un algoritmo genético dé buenos resultados.

Como método general de resolución de problemas, los algoritmos genéticos también se pueden utilizar para resolver problemas de clasificación. Es el caso, por ejemplo, de los sistemas clasificadores [*classifier systems*]. En los sistemas clasificadores de tipo Michigan, propuestos por el propio John Holland y desarrollados más tarde por Stewart Wilson, cada individuo representa una regla y la solución es el conjunto de individuos de la población. En los sistemas clasificadores de tipo Pittsburgh, propuestos por Stephen Smith bajo la supervisión de Kenneth De Jong, cada individuo representa una solución completa, un conjunto de reglas de longitud variable. Los clasificadores construidos utilizando algoritmos genéticos suelen destacar por ser relativamente robustos: su rendimiento no se ve excesivamente afectado por la aparición de errores en los casos de entrenamiento, algo que sí puede ocurrir con otros modelos simbólicos más tradicionales. Sin embargo, su coste computacional no siempre los hace adecuados para problemas de minería de datos, especialmente en *big data*.

Programación evolutiva

Lawrence J. Fogel es el padre de la programación evolutiva: el uso de la evolución simulada para resolver problemas reales. La evolución simulada se interpreta como un proceso de aprendizaje que conduce a la consecución de “inteligencia artificial”, que Fogel entendía como comportamiento adaptativo, sin más. Ahora bien, para conseguir un comportamiento adaptativo, la realización de predicciones se considera un requisito previo: si no sabemos qué efectos puede tener algo, tampoco podemos determinar en qué sentido debemos cambiar nuestro comportamiento. En consecuencia, Fogel llegó a la conclusión de que la capacidad de realizar predicciones es clave para la inteligencia.

En la propuesta original de Fogel, la estructura del “programa” que

Además, Lawrence J. Fogel realizó la primera tesis doctoral en computación evolutiva (“On the Organization of Intellect”, UCLA, 1964) y publicó el primer libro de computación evolutiva (con Alvin Owens y Michael Walsh: “Artificial Intelligence through Simulated Evolution”, Wiley, 1966).

se pretende optimizar se mantiene fija, en forma de autómata finito. Son los parámetros de ese programa los que pueden evolucionar. Propuestas posteriores sí que permiten que la propia estructura del programa evolucione, sin tener que mantener fija dicha estructura.

Un algoritmo de programación evolutiva genera soluciones iterativamente, con la intención de que cada vez sean más aptas para su entorno (lo que se evalúa mediante una función de *fitness*, como en los algoritmos genéticos). El tamaño de la población utilizada puede variar dependiendo del problema y de la capacidad de cálculo disponible (recordemos que estamos hablando de una técnica desarrollada en los años 60 del siglo XX, por lo que había que aprovechar al máximo los escasos recursos de los que se disponía entonces si los comparamos con los ordenadores actuales). Durante la evolución de la población, cada parente da lugar directamente a varios hijos, sin los mecanismos de reproducción sexual de los algoritmos genéticos. Para mantener la diversidad de la población, se ajusta la probabilidad de cada tipo de mutación y el número de mutaciones por hijo.

El éxito más conocido de la programación evolutiva es el juego de damas Blondie24, desarrollado por David B. Fogel, el hijo de William. Blondie24 utiliza un algoritmo evolutivo para ajustar los 5046 pesos de la red neuronal que evalúa el valor de un movimiento en el juego de las damas. Distintos programas, con sus respectivas redes neuronales, juegan entre sí sin intervención humana ni ningún tipo de conocimiento específico del juego. Tras 840 generaciones, que se tradujeron en 6 meses de calendario, la mejor estrategia obtenida se probó con jugadores humanos por Internet en 1999. Blondie24 fue capaz de superar en el ranking al 99.61 % de los jugadores humanos.

Posteriormente, David Fogel desarrolló un programa evolutivo para jugar al ajedrez, Blondie25. Su programa, que aprendió a jugar al ajedrez sin intervención humana, fue capaz de vencer tres veces a Fritz 8 (el quinto mejor programa de ajedrez del mundo en ese momento) de 24 partidas que disputaron, con 11 tablas y 10 victorias para Fritz. También fue la primera máquina de su tipo capaz de vencer a un maestro humano, James Quon, ganando tres de las cuatro partidas que disputaron en 2006. No está mal para una máquina que aprende a jugar al ajedrez desde cero, sin el conocimiento experto integrado en sofisticados programas de ajedrez como Deep Blue o el propio Fritz.

Estrategias de evolución

Ingo Rechenberg, Hans-Paul Schwefel y, más tarde, Peter Bienert, desarrollaron las estrategias de evolución [*ES: Evolution Strategies*].

Las estrategias de evolución se basan en realizar ajustes discretos aleatorios y están inspiradas en las mutaciones que ocurren en la naturaleza.

La versión original de las estrategias de evolución, de 1971, usa un solo parental del que se genera un único hijo, motivo por el que se denomina (1+1)-ES. El hijo se mantiene si es mejor que el parental. De lo contrario, se elimina. De ahí proviene el uso del signo más en (1+1)-ES: la selección se realiza considerando tanto al parental como al hijo.

El mecanismo de selección utilizado por las estrategias de evolución es determinista, a diferencia de los algoritmos genéticos, que realizan la selección de forma probabilística teniendo en cuenta los valores de la función de *fitness*. En la selección extintiva utilizada por las estrategias de evolución, los peores individuos tienen una probabilidad cero de ser seleccionados.

Ingo Rechenberg, en 1973, introdujo el concepto de población en sus estrategias evolutivas. A partir de una población de tamaño μ , se genera un único hijo, que se crea a partir de uno de los μ posibles padres. El parental se elige aleatoriamente de entre los miembros de la población, pero la selección sigue siendo extintiva: las estrategias evolutivas ($\mu+1$)-ES utilizan al hijo creado para reemplazar al peor parental de la población.

Un par de años después, en 1975, Hans-Paul Schwefel añadió la posibilidad de crear múltiples hijos por generación. A partir de una población de μ individuos, se crean λ hijos en una misma generación. Este cambio permite la aparición de dos variantes a la hora de realizar la selección:

- Estrategias de evolución ($\mu+\lambda$)-ES: Los μ mejores individuos del conjunto de padres e hijos sobreviven; esto es, se junta el conjunto de padres con el conjunto de hijos y se descartan los peores individuos, como en las estrategias (1+1)-ES y ($\mu+1$)-ES.
- Estrategias de evolución (μ, λ)-ES: Sólo los μ mejores hijos sobreviven hasta la siguiente generación. Los padres se descartan automáticamente y nunca sobreviven (como en los algoritmos genéticos, salvo que éstos utilicen algún mecanismo de selección elitista que siempre conserve a los mejores individuos). Para mantener la presión selectiva elevada, se crea un número de hijos λ mucho mayor que el número de individuos μ de la población, p.ej. $\lambda \approx 7\mu$.

Programación genética

Los primeros intentos de hacer que un programa de ordenador fuese capaz de evolucionar se remontan hasta los años 50 y 60 del siglo XX. No obstante, hasta los años 80 no se obtuvieron los primeros resultados satisfactorios. Joseph Hicklin y Cory Fujiki, como estudiantes de posgrado en la Universidad de Idaho, usaron expresiones-S en LISP para representar programas cuyo objetivo era resolver problemas de teoría de juegos. Nichael Lynn Cramer de Texas Instruments, en 1985, y John R. Koza de la Universidad de Stanford, en 1989, propusieron de forma independiente

el uso de una representación en forma de árbol sobre la que se implementa un operador de cruce que permite intercambiar subárboles entre los diferentes programas de una población generada al azar.

La propuesta de Koza fue la que se acabó imponiendo y, más tarde, se denominó programación genética. En realidad, no es más que un algoritmo genético que funciona sobre árboles, en vez de sobre cadenas de bits o vectores de números.

La programación genética, como una forma más de programación automática, fue considerada por muchos una moda pasajera en Inteligencia Artificial, un intento más de lograr lo imposible.

En 2004, Koza creó los premios *Humie* que anualmente se otorgan para reconocer las creaciones genéticas que resultan competitivas con los mejores diseños realizados por el hombre. Se han otorgado decenas de ellos: <http://www.human-competitive.org/>

Técnicas conexionistas: I.A. conexionista

Llegamos, por fin, a la quinta familia de técnicas de aprendizaje automático. La que, por ahora, se ha acercado más al imposible de lograr máquinas capaces de programarse a sí mismas. Si bien no en el sentido general de la programación genética de John Koza, sí en el sentido más restringido de la programación evolutiva de William y David Fogel.

Esta familia, obviamente, es la de los investigadores que emplean redes neuronales artificiales⁷⁵ en la resolución de problemas de aprendizaje automático. Desde los albores de la Inteligencia Artificial, las redes neuronales se utilizaron como modelo para intentar dotar de inteligencia a las máquinas. Los conexionistas, como se les llamaba a los que utilizaban el cerebro como modelo, proponían un enfoque alternativo a las técnicas basadas en modelos lógicos de los simbólicos. De ahí que las redes neuronales también recibiesen la denominación de técnicas subsimbólicas (no sin cierto desdén, según se pronuncie).

Actualmente, se hace referencia a las redes neuronales artificiales con el término *deep learning*.⁷⁶ Literalmente, aprendizaje profundo. No obstante, dado que la traducción literal del término puede hacer pensar que estamos hablando de una trama de informadores en una película de cine negro, mantendremos el anglicismo para hacer referencia a las técnicas desarrolladas para diseñar y entrenar redes neuronales artificiales.

Tras distintos altibajos desde sus orígenes, las redes neuronales artificiales se han convertido en las técnicas de moda dentro del aprendizaje automático. ¿Por qué? Porque hasta ahora, han sido las únicas capaces de alcanzar una eficacia comparable con la del hombre en distintos problemas complejos de reconocimiento de patrones, como el reconocimiento de voz (utilizado hoy por numerosos dispositivos móviles y gadgets particulares), la traducción de textos de un idioma a otro, o la identificación de objetos en imágenes, un problema típico de visión artificial.

Fue precisamente en un congreso de visión artificial, el CVPR [*Conference on Computer Vision and Pattern Recognition*], donde uno de los investigadores de mayor prestigio en redes neuronales artificiales, el

⁷⁵ Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1996. ISBN 0198538642

⁷⁶ Ian Goodfellow, Yoshua Bengio, y Aaron Courville. *Deep Learning*. MIT Press, 2016. ISBN 0262035618. URL <http://www.deeplearningbook.org>

francés Yann LeCun protagonizó un pequeño incidente en 2012, similar en su naturaleza al de John Koza en el ICML años atrás. Al enviar sus trabajos a congresos especializados en un tema particular, como el caso de CVPR, no es extraño que un autor que utilice técnicas poco convencionales en el área se encuentre con comentarios muy críticos hacia su trabajo. Tras recibir tres evaluaciones negativas para un artículo suyo de 2012 (“*definitely reject*”, “*weakly reject*”, “*borderline*”), algo que no era la primera vez que sucedía, decidió retirarlo del congreso y enviarle una carta al organizador, que posteriormente se hizo pública. Las revisiones recibidas eran excesivamente parciales y mostraban la aversión de los revisores [*referees*] hacia las redes neuronales, cuando no su ignorancia acerca de ellas. En palabras del propio LeCun: “*the reviews [are] so ridiculous, that I don't know how to begin writing a rebuttal without insulting the reviewers*” (las revisiones son tan ridículas, que no sé cómo comenzar a escribir una refutación sin insultar a los revisores). Aun reconociendo el nivel de exigencia de un congreso de prestigio como el CVPR, LeCun decidió no enviar más artículos sobre redes neuronales al CVPR (¡un congreso que él mismo había organizado años atrás!). Una muestra más de los problemas sociológicos asociados a la formación de escuelas dentro de un área de conocimiento, que pueden llegar a comportarse como auténticas sectas. Cansado de desperdiciar su tiempo y esfuerzo en intentar colocar sus trabajos en el CVPR, donde sus trabajos sobre redes neuronales se recibían con una hostilidad manifiesta, LeCun se despidió en su carta esperando volver a encontrarse con el organizador en alguno de los dos congresos más importantes donde sí se publicaban artículos sobre redes neuronales artificiales: NIPS [*Conference on Neural Information Processing Systems*], centrada precisamente en redes neuronales, e ICML [*International Conference on Machine Learning*], sobre técnicas de aprendizaje automático en general.

Afortunadamente, el encontronazo entre los investigadores en redes neuronales artificiales y los especialistas en visión artificial no duró demasiado. En octubre de 2012, un equipo de la Universidad de Toronto liderado por Geoffrey Hinton y compuesto por Alex Krizhevsky e Ilya Sutskever que utilizaba redes neuronales para identificar objetos en imágenes venció claramente a las técnicas tradicionales de visión artificial en la competición ILSVRC [*ImageNet Large Scale Visual Recognition Challenge*].⁷⁷ Algo que las redes neuronales artificiales han repetido desde entonces en todas las ediciones de la competición. A los especialistas en visión artificial no les quedó más remedio que rendirse ante la evidencia. Su actitud hacia las redes neuronales artificiales ha cambiado y ahora son ellos los que también trabajan en redes neuronales artificiales. Numerosos artículos del CVPR tratan sobre redes neuronales y el propio LeCun ha sido invitado para dar charlas en un congreso en el que estuvo a punto de tirar la toalla, por lo desmoralizadoras que resultaban, para los miembros

Yann LeCun es conocido por el desarrollo de redes neuronales convolutivas para resolver problemas de reconocimiento óptico de caracteres [*OCR: Optical Character Recognition*], como las que diseñó en su etapa en los laboratorios de investigación de AT&T y luego se utilizaron para procesar millones de cheques en Estados Unidos, donde es habitual pagar los recibos de servicios básicos de suministro [*utilities*], como agua o electricidad, enviando un cheque por correo. Fuera del ámbito de la I.A., también fue uno de los creadores de la tecnología de compresión de imágenes utilizada por el formato DjVu, una alternativa al omnipresente PDF de Adobe. Actualmente, dirige el laboratorio de investigación en I.A. de Facebook en Nueva York.

El artículo de LeCun describía una primera versión de un sistema de segmentación semántica de imágenes (etiquetado de escenas) basado en redes neuronales convolutivas. Su sistema mejoraba el estado de arte de entonces, evaluándolo sobre tres conjuntos de datos estándar, y era un orden de magnitud más rápido que su competidor más cercano. Además, era completamente automático y no requería el diseño manual de características a medida para el problema (algo que uno de los revisores del CVPR consideraba... ¡negativo!). El artículo se publicó inicialmente en arXiv, un servicio de publicación de resultados de investigación muy popular. Ese mismo año, fue aceptado en otro congreso de primer nivel, el ICML (el del incidente del lago Tahoe), donde los revisores se mostraron más receptivos hacia las propuestas que se presentaban. Posteriormente, una versión extendida del artículo aparecería publicada en IEEE Transactions on Pattern Analysis and Machine Intelligence, una de las revistas más prestigiosas del área.

⁷⁷ Alex Krizhevsky, Ilya Sutskever, y Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. En *NIPS'2012 Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL <https://goo.gl/Ddt8Jb>

de su equipo, las críticas injustificadas hacia los métodos que utilizaban.

La idea común detrás de todas las técnicas basadas en redes neuronales artificiales es intentar expresar la solución de problemas complejos no como un algoritmo secuencial que resuelve un problema aplicando una secuencia de pasos concreta, sino como el resultado de combinar, en paralelo, las pequeñas contribuciones realizadas por un gran número de elementos simples de procesamiento que se hallan interconectados entre sí. Esos elementos simples de procesamiento son las neuronas que forman parte de una red neuronal artificial y las conexiones entre ellas son las que representan el “conocimiento” adquirido por la red.

Las redes neuronales artificiales, obviamente, se inspiran en el funcionamiento del cerebro humano. En ocasiones, intentan modelar fielmente alguno de los procesos que realiza nuestro cerebro. Otras veces, sin embargo, hacen una interpretación más libre de cómo se podría resolver un problema usando un sistema distribuido con múltiples unidades de procesamiento, aunque esa forma de resolver el problema no resulte plausible desde el punto de vista biológico.

Los conexionistas más cercanos a las ciencias biológicas basan su trabajo en los descubrimientos que la neurociencia nos ha proporcionado en su intento de hacer ingeniería inversa del funcionamiento del cerebro humano. Se intentan mantener fieles a la biofísica de las neuronas biológicas e intentan desarrollar sistemas neuromórficos. Ven las redes neuronales artificiales como una de las aportaciones más importantes que las ciencias experimentales han realizado al campo de la Inteligencia Artificial.

Los conexionistas más vinculados al desarrollo de técnicas de Inteligencia Artificial se suelen centrar en el diseño de algoritmos de aprendizaje. Desde el punto de vista formal, una sencilla red neuronal multicapa se comporta como un aproximador universal. Esto es, es capaz de aprender cualquier cosa que se pueda aprender a partir de un conjunto de datos de entrenamiento dado.

El entrenamiento de una red neuronal se realiza con algoritmos como *backpropagation*. Este algoritmo, de propagación de errores hacia atrás, utiliza el error observado en la salida de una red neuronal multicapa para ir ajustando paulatinamente los parámetros que definen la respuesta de la red ante una entrada dada, con la intención de reducir el error cometido por la red.

Las redes neuronales son, pues, capaces de aprender a partir de ejemplos, lo que les permite generalizar sin tener que formalizar el conocimiento adquirido. Por este motivo, se comportan como cajas negras, difíciles de interpretar para nosotros. De ahí lo de referirse a ellas como modelos subsimbólicos. Al resultar totalmente incomprensibles, incluso para un experto humano, es más difícil que los modelos basados en redes neuro-

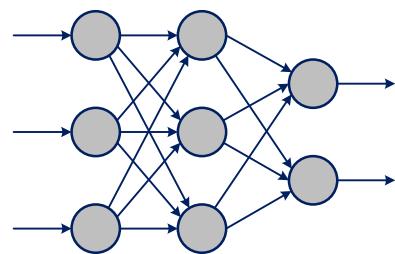


Figura 20: Una red neuronal artificial con múltiples capas de neuronas. La capa de entrada, a la izquierda, recibe un estímulo del exterior. Esa entrada se acaba traduciendo en una respuesta en la capa de salida, a la derecha. Entre las capas de entrada y salida pueden existir capas intermedias que se denominan capas ocultas.

Si aceptamos pulpo como animal de compañía y somos flexibles en lo que consideramos una red neuronal, las técnicas de aprendizaje por cuantificación vectorial de Teuvo Kohonen, LVQ [*Linear Vector Quantization*], serían una honrosa excepción en cuanto a la interpretabilidad de las redes neuronales, ya que los algoritmos LVQ destacan por la sencillez de las heurísticas que utilizan.

nales sean aceptados en determinados contextos frente a otros modelos más interpretables, como los modelos simbólicos basados en árboles o reglas. Dicho de otro modo, las redes neuronales pueden resultar ideales como modelo predictivo, pero no suelen resultar tan útiles como modelos descriptivos.

Conforme vayamos profundizando en el tema, nos iremos encontrando con algunos investigadores destacados que han realizado contribuciones notables en el ámbito de las redes neuronales artificiales. Con algunos de ellos, de hecho, ya nos hemos cruzado. Si hablamos de *deep learning*, los tres pioneros que desencadenaron la fiebre actual por las redes neuronales artificiales son un inglés y dos franceses (y no es un chiste):

- Geoffrey Everest Hinton, a.k.a. Geoff Hinton, de la Universidad de Toronto en Canadá y Google: Un inglés con nacionalidad canadiense que es, además, tataranieto del matemático lógico George Boole (sí, el de la lógica booleana).
- Yann LeCun, de la Universidad de Nueva York (NYU) y Facebook AI Research (FAIR): Tras realizar su doctorado en París, realizó una estancia posdoctoral en el laboratorio de Hinton en Toronto, tras la que pasó a trabajar en los Laboratorios Bell de AT&T en Holmdel, Nueva Jersey.
- Yoshua Bengio, de la Universidad de Montréal, también en Canadá: Nacido en Francia, se doctoró en la Universidad McGill de Montréal, trabajó en el MIT bajo la supervisión de Michael Jordan y también pasó por los Laboratorios Bell de AT&T.

Como tendremos ocasión de ver, los tres publicaron artículos en 2006 que sirvieron para relanzar la investigación en redes neuronales artificiales y revitalizar el campo que hoy conocemos como *deep learning*.

Para resolver un problema de aprendizaje automático, primero se diseña una red neuronal estableciendo su topología (su estructura) y luego se entrena utilizando un algoritmo como *backpropagation*. El secreto suele estar en encontrar cuál es la topología ideal para resolver el problema concreto al que nos estemos enfrentando: ¿qué tipo de red neuronal? ¿con cuántas capas? ¿con qué tipo de neuronas en cada capa?... Una vez establecidos los parámetros de la red, hay que elegir los parámetros del algoritmo de aprendizaje: ¿qué estrategia de ajuste de parámetros? ¿qué algoritmo de optimización? ¿qué mecanismos de regularización?... Un sinfín de hiperparámetros que tendremos que ajustar de forma heurística y para lo que analizaremos con detalle cada uno de los aspectos que intervienen en el entrenamiento de una red neuronal. A menudo, un arte más que una ciencia.

Desde el punto de vista de la descomposición del error en sesgo y varianza, nuestros modelos basados en redes neuronales nos obligarán a

alcanzar un compromiso. Si utilizamos redes neuronales pequeñas, éstas tendrán menos parámetros que ajustar, por lo que su entrenamiento será más sencillo desde el punto de vista computacional. No obstante, un número reducido de parámetros puede inducir un sesgo en nuestro modelo, haciéndolo propenso al infraaprendizaje [*underfitting*] si el número de parámetros no es suficiente para modelar la complejidad del problema que estemos resolviendo. Si, por el contrario, optamos por construir una red neuronal grande, ésta incluirá un número elevado de parámetros, lo que hará mucho más costoso computacionalmente su entrenamiento. Además, cuantos más parámetros tenga la red, más propensa será al sobreaprendizaje [*overfitting*], lo que intentaremos evitar utilizando mecanismos diseñados específicamente para prevenir el sobreaprendizaje: los mecanismos de regularización.

Pero, antes de entrar en detalle, aún nos faltan por ver un par de temas relevantes relacionados con las técnicas de aprendizaje automático: cómo combinar múltiples modelos y cómo preparar los datos para suministrárselos de entrada a los algoritmos de entrenamiento con los que construiremos nuestros modelos.

Combinación de múltiples modelos: Ensembles

En aprendizaje automático, un ensemble no tiene nada que ver con un cuarteto de cuerda. O tal vez sí, aunque sea en sentido figurado. Un ensemble hace referencia a una colección de modelos que se utilizan conjuntamente a la hora de realizar una predicción. Para un director de orquesta, que un músico desafine puede resultarle bastante molesto. Para un científico de datos, que un modelo se equivoque es un hecho inevitable que ha de asumir. Cuando se juntan varios músicos que desafinan y, además, lo hacen en segmentos diferentes de su interpretación, el director de orquesta vive una situación desesperante, una pesadilla para él. Cuando son varios modelos los que se equivocan, pero lo hacen en situaciones diferentes, es casi el paraíso para un científico de datos. ¿Por qué esa diferencia? Porque los errores de unos se pueden compensar con los aciertos de los demás modelos del ensemble y éste, en conjunto, obtendrá mejores resultados que cualquiera de sus miembros por separado. En ese sentido, un ensemble de modelos no se parece en nada a la cacofonía musical de un ensemble musical en el que cada músico desafina por separado, por suerte para nosotros frente a los desdichados directores de orquesta.

Algunos, de forma algo exagerada, hablan de la sabiduría de las multitudes [*wisdom of crowds*]. En 1906, el prolífico estadístico inglés Francis Galton visitó una feria de ganado en Plymouth. En esa feria, los visitantes eran invitados a adivinar el peso de un buey y Galton pudo analizar las 800 estimaciones del peso del animal. Las estimaciones

individuales variaban bastante, por encima o por debajo, como era de esperar, pero Galton observó que la mediana de éstas era de 1,207 libras (unos 547 kilogramos). Sorprendentemente cerca de las 1,197 libras del peso real del animal. ¡Menos de un 1% de error! La media de las estimaciones, de hecho, era de 1,197 libras. ¡El peso exacto! La multitud era capaz de estimar el peso real, aun cuando cada estimación, por separado, podía incurrir en errores significativos.

Obviamente, un resultado así sólo es válido si las estimaciones se realizan de forma independiente. Si se realizasen de forma secuencial y pública, por ejemplo, cada estimador conocería las estimaciones anteriores. En tal situación, las estimaciones ya realizadas sesgarían las estimaciones posteriores. Se podrían producir cascadas de información [*information cascades*]: una persona observa las acciones de otras y, aunque entre en contradicción con su información privada (su estimación a priori), se deja llevar por las opiniones de los demás. Uno más de los muchos sesgos psicológicos que nos afligen a la hora de tomar decisiones.

Por otro lado, hay que tener en cuenta que no todas las cuestiones pueden resolverse recurriendo a la “sabiduría” de la gente. Si sometiésemos al Parlamento la aprobación de la ley de la gravitación universal formulada por Isaac Newton en 1687, se podría criticar que no es lo suficientemente igualitaria, pues discrimina a los individuos con sobrepeso. Tal vez incluso sería rechazada, especialmente si la argumentación se adereza con ataques ad hominem sobre si Newton era de derechas o de izquierdas. Ahora bien, una decisión de espaldas a la realidad, por muy democrática que sea, siempre resultará equivocada. Algo que en Economía sucede continuamente.

El aprendizaje automático, por norma general, se aplica allí donde no existen aún leyes científicamente contrastadas. Por tanto, si somos capaces de combinar varios modelos adecuadamente, podremos mejorar el rendimiento de cada uno de esos modelos por separado. En realidad, los ensambles funcionan tan bien, que son los que suelen ganar las competiciones que se realizan en sitios como Kaggle. El premio Netflix de un millón de dólares, por ejemplo, se lo llevó un equipo que utilizaba un ensemble de sistemas de recomendación. El equipo ganador en sí, *BellKor's Pragmatic Chaos*, era a su vez un ensemble de tres equipos que habían comenzado su andadura por separado en la competición organizada por Netflix. Su solución combinaba unos 500 predictores de todo tipo, como árboles de decisión, clasificadores basados en casos y redes neuronales.

¿Cómo podemos combinar múltiples modelos? El ensemble ha de realizar una predicción a partir de las predicciones efectuadas por los modelos independientes. Como tal, el ensemble no construye un modelo del problema que estamos resolviendo, sino que aplica un algoritmo para agregar la información proporcionada por los modelos individuales, por

Aun cuando la media aritmética ofrecía una mejor estimación, Galton defendía el uso de la mediana frente a distintos tipos de medias cuando un jurado tuviese que estimar daños o un comité tuviese que conceder becas. Lo veía más democrático, correspondiente al voto del 50% + 1.

Otros de los sesgos psicológicos más comunes son los que los psicólogos denominan: efecto marco [*framing*] y anclaje [*anchoring*]. Una revista ofrece una suscripción a su edición online por \$59 y a su edición impresa por \$125, junto con una oferta combinada que ofrece tanto la edición impresa como la edición online por \$125, el mismo precio que la edición impresa. ¿Qué sentido tiene? Establecer un contexto. Si sólo se ofreciesen dos opciones, muchos optaríaían por la suscripción más económica. Al ofrecerse una tercera, la mayoría se decantará por la oferta (y nadie se quedará con la segunda opción). Cuando uno visita una tienda de colchones, se encuentra incentivos diseñados para influir en su comportamiento: los colchones están permanentemente en oferta para crear en sus clientes la necesidad acuciante de tomar ya una decisión de compra. Aunque el ahorro sea completamente virtual, ¿quién va a renunciar a ahorrarse unos cientos de euros o dólares? En casos así, somos predeciblemente irracionales.

Dan Ariely. *Predictably Irrational: The Hidden Forces That Shape Our Decisions*. Harper, 2009. ISBN 0061854549

lo que podríamos decir que es un metamodelo. Más o menos, como un político que ha de tomar decisiones sobre temas que desconoce a partir de la opinión de sus asesores. Pero, afortunadamente, sin los conflictos de interés e incentivos perversos que podrían afectar a la toma de decisiones políticas.

La construcción de un ensemble es un problema de aprendizaje supervisado. Podemos aplicar técnicas de aprendizaje supervisado para resolver el problema de la combinación de opiniones, igual que cada modelo individual las aplica sobre el problema original. Es el denominado meta-aprendizaje: aprender a partir de los modelos de aprendizaje. El modelo derivado de ese meta-aprendizaje puede ser de cualquier tipo, desde una simple votación ponderada hasta un árbol de decisión o una red neuronal.

Los algoritmos de aprendizaje son estocásticos por naturaleza y su comportamiento puede variar sobre el mismo conjunto de datos. Los métodos de remuestreo, como la validación cruzada, sirven para evaluar cuánta varianza puede existir en los resultados obtenidos por un método concreto de aprendizaje. En términos de sesgo y varianza, los ensembles nos ayudan a reducir la varianza. Además esa reducción de varianza puede venir acompañada por la anulación del sesgo propio de modelos individuales, lo que explica por qué un ensemble puede ser mejor que cualquiera de los modelos individuales que lo componen.

Empíricamente, los ensembles obtienen mejores resultados cuando existe cierta diversidad en los modelos que los componen. Se pueden emplear diferentes estrategias para introducir esa diversidad en un ensemble particular. Las más conocidas responden a los nombres de *bagging*, *boosting* y *stacking*.

Bagging

Cuando los ensembles se construyen usando *bagging*, varios clasificadores diferentes votan para decidir la clase de un caso particular y la votación se realiza por mayoría simple.

Para construir los clasificadores del ensemble, se utiliza *bootstrapping*, de donde proviene el nombre de la técnica: *bagging* como abreviatura de *bootstrap aggregating* (esto es, agregación con *bootstrapping*). Se crean numerosas versiones del conjunto de entrenamiento utilizando muestreo con reemplazo y cada una de esas versiones se utiliza para construir un modelo de aprendizaje diferente. Los distintos modelos así obtenidos se combinan en un ensemble y, para realizar predicciones, las predicciones individuales se promedian.

Bagging suele construir modelos homogéneos, todos del mismo tipo, aunque entrenados sobre diferentes muestras del conjunto de datos original. Se aprovecha del hecho de que las diferencias en los conjuntos de

entrenamiento individuales introducirán cierta variedad en los modelos construidos. Puede parecer que la construcción de múltiples modelos introduce un grado mayor de flexibilidad que podría conducir a sobreajustar los datos de entrenamiento, el temido sobreaprendizaje [*overfitting*]. Sin embargo, al agregar los resultados individuales, el ensemble reduce la varianza asociada a la técnica de aprendizaje concreta que se emplee para construir los modelos individuales.

El estadístico Leo Breiman, de la Universidad de Berkeley, introdujo esta técnica en 1996.⁷⁸ Cuando los modelos individuales son árboles de decisión que, además, se entranan con subconjuntos aleatorios de características para aumentar aún más su variabilidad, el resultado se denomina *random forest* (literalmente, árbol aleatorio). Los *random forests* fueron descritos por Leo Breiman en 2001.⁷⁹

¿Por qué se entrena cada árbol sobre un subconjunto distinto de características? Porque, al introducir aleatoriedad, se obtiene un ensemble más robusto que si utilizásemos un algoritmo que construyese siempre los árboles utilizando todos los atributos de los que dispongamos.

Los *random forests* se aprovechan de que los algoritmos de construcción de árboles de decisión son eficientes y escalables. Se construyen múltiples árboles de decisión de forma independiente, lo que se puede hacer en paralelo. Construyendo árboles CART [*Classification And Regression Trees*], podemos resolver problemas tanto de clasificación como de regresión. El resultado proporcionado por el ensemble será la moda o la media de la predicción efectuada por los árboles de decisión individuales, dependiendo de si estamos ante un problema de clasificación o ante un problema de regresión.

El sesgo del *random forest* es equivalente al de un árbol de decisión individual (que tiene una varianza elevada). Creando muchos árboles (esto es, un bosque) y agregando sus predicciones, la varianza del modelo se reduce con respecto a la varianza de un árbol individual. Cuantos más árboles contenga el bosque, más se reducirá la varianza sin aumentar el sesgo del modelo. En la práctica, el tamaño del bosque vendrá limitado por los recursos computacionales de los que dispongamos, ya que no podemos entrenar un número infinito de árboles de decisión.

Boosting

El *boosting* fue creado por Yoav Freund y Rob Schapire, dos investigadores de los laboratorios de AT&T en Nueva Jersey, en 1996.⁸⁰ Mientras que en *bagging* se realiza una votación por mayoría simple en la que el voto de cada modelo del ensemble tiene el mismo peso, en *boosting* se realiza una votación ponderada. Los clasificadores que forman parte del ensemble tienen distintos pesos en la votación dependiendo de su precisión.

⁷⁸ Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, August 1996. ISSN 0885-6125. DOI: 10.1023/A:1018054314350

⁷⁹ Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, October 2001. ISSN 0885-6125. DOI: 10.1023/A:1010933404324

Random Forests® es una marca registrada por Leo Breiman y Adele Cutler, que vendieron su licencia en exclusiva a Salford Systems, una empresa dedicada al desarrollo de herramientas de minería de datos: <https://www.salford-systems.com/>

⁸⁰ Yoav Freund y Robert E. Schapire. Experiments with a new boosting algorithm. En Lorenza Saitta, editor, *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96)*, Bari, Italy, July 3–6, 1996, pages 148–156. Morgan Kaufmann, 1996. ISBN 1-55860-419-7

En lugar de combinar diferentes algoritmos de aprendizaje, *boosting* funciona aplicando el mismo clasificador a los datos de forma repetida. Cada nuevo modelo se utiliza para intentar corregir los errores cometidos por los anteriores modelos. ¿Cómo? Asignando pesos a los ejemplos de entrenamiento. El peso de un ejemplo del conjunto de entrenamiento aumenta cuando se clasifica mal, por lo que se considerará más importante en las siguientes iteraciones.

El nombre de la técnica, *boosting*, que se podría traducir por ‘impulsar’, hace referencia a que el proceso iterativo permite impulsar de forma consistente un modelo de clasificación inicial, que sólo sea ligeramente mejor que una predicción aleatoria, hasta obtener un clasificador casi perfecto. Esto es, es una estrategia que permite construir clasificadores fuertes a partir de clasificadores débiles. Además, lo consigue hacer de forma eficiente.

En la votación, los modelos que suelen predecir la clase correcta tendrán un peso elevado, mientras que los más imprecisos tenderán a ser ignorados.

Boosting, como técnica de construcción de ensambles, se centra principalmente en reducir el sesgo de los clasificadores que forman el ensemble. En algunos casos, *boosting* consigue mejores resultados que *bagging* pero también tiende a producir modelos que sobreajusten el conjunto de entrenamiento (*bagging* evita ese sobreajuste al centrarse en reducir la varianza).

La implementación más habitual de *boosting* es el algoritmo AdaBoost, la propuesta de Freund y Schapire por la que recibieron el premio Gödel en 2003. AdaBoost proviene de *adaptive boosting*.

Gradient boosting consiste en interpretar el *boosting* como un proceso de optimización sobre una función de coste diferenciable, también conocida como función de pérdida [*loss function*]. Los algoritmos de *gradient boosting* son algoritmos iterativos que optimizan la función de coste eligiendo una hipótesis que apunte en la dirección del gradiente descendente, la misma idea que se utiliza para entrenar redes neuronales. Es habitual utilizar esta estrategia utilizando árboles de decisión, como sucede en la biblioteca XGBoost, disponible para los lenguajes de programación C++, Java, Python, R y Julia.

Stacking

Como estrategia alternativa para la construcción de ensambles mencionaremos, por último, al *stacking* o “generalización apilada” [*stacking generalization*].⁸¹ Fue propuesta originalmente por David Wolpert en 1992 (sí, otra vez, el mismo del nada es gratis en las técnicas de aprendizaje supervisado). La estrategia consiste en entrenar un algoritmo de aprendizaje para combinar las predicciones de otros algoritmos de aprendizaje.

⁸¹ David H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, February 1992. ISSN 0893-6080. DOI: 10.1016/S0893-6080(05)80023-1

En primer lugar, se entrenan múltiples modelos predictivos utilizando los datos disponibles. A continuación, se entrena un algoritmo de combinación que sea capaz de realizar una predicción final a partir de las predicciones de los modelos previamente entrenados, utilizando sus predicciones como entradas.

Si se admite cualquier algoritmo de combinación, el *stacking* engloba a las demás técnicas de construcción de ensambles, incluidos *bagging* y *boosting*. No obstante, lo más habitual es utilizar un modelo de regresión logística como algoritmo de combinación de las predicciones individuales de los distintos miembros del ensemble.

Se ha utilizado con éxito en problemas de clasificación, regresión y también para resolver problemas de aprendizaje no supervisado. Incluso se ha empleado para estimar el error asociado al uso de *bagging*. Los dos equipos finalistas de la competición del millón de dólares de Netflix utilizaron *blending*,⁸² que puede verse como una forma de *stacking*.

Selección y extracción de características

Uno de los problemas más habituales en muchas técnicas de aprendizaje automático es la maldición de la dimensionalidad [*curse of dimensionality*], una expresión acuñada por el matemático Richard Bellman cuando desarrollaba las técnicas de optimización conocidas con el término, algo engañoso, de programación dinámica. Bellman observó que las técnicas de control que funcionaban bien en dos o tres dimensiones eran excesivamente ineficientes cuando aumentaba el número de variables que deseaba controlar; esto es, en espacios con un número mayor de dimensiones.

¿Por qué es un problema la dimensionalidad de los datos? Principalmente, porque conforme aumentamos el número de dimensiones, el volumen del espacio multidimensional aumenta exponencialmente, mientras que el conjunto de datos del que disponemos no lo hace. Como consecuencia, dispongamos de los datos de los que dispongamos, el resultado final es un conjunto de datos muy disperso [*sparse*] y esa dispersión complica su análisis desde el punto de vista estadístico.

Por poner un ejemplo concreto, si tenemos datos personales sobre la edad y nivel de renta de los individuos de una población, podremos utilizar técnicas de agrupamiento o segmentación para identificar segmentos demográficos sin más que medir las distancias de un punto a otro en el espacio bidimensional definido por nuestras dos variables (edad y nivel de renta). Cuando añadimos una tercera dimensión, como el nivel educativo de cada persona, muestras que en dos dimensiones estaban cerca puede que queden algo más lejos. Si seguimos añadiendo una cuarta, quinta o septingentésima dimensión, las instancias que inicialmente eran parecidas estarán cada vez más lejos unas de otras en ese espacio n-dimensional,

⁸² Joseph Sill, Gábor Takács, Lester W. Mackey, y David Lin. Feature-weighted linear stacking. *arXiv e-prints*, 2009. URL <http://arxiv.org/abs/0911.0460>

Tal como reconoce el propio Bellman en su autobiografía, la denominación de “programación dinámica” no fue más que una táctica de marketing. En una época hostil a la investigación matemática, el no va más de la tecnología en los años 50 eran los primeros ordenadores electrónicos. Por ese motivo, Bellman, que trabajaba en técnicas de optimización para resolver problemas de planificación temporal, buscó un nombre llamativo que le permitiese obtener financiación para sus proyectos. La programación era algo muy novedoso por entonces y el adjetivo dinámico carecía de connotaciones peyorativas. El resultado, algo a lo que ni siquiera un congresista pudiese ponerle peros. De haberlo hecho hoy en día, la técnica igual se llamaría desarrollo sostenible (o *deep learning*).

hasta llegar un momento en el que todas estén lejos unas de otras. Cuando tenemos muchas dimensiones, las medidas de distancia dejan de ser útiles debido a la equidistancia de unas muestras a otras, lo que nos impide establecer comparaciones de forma adecuada. De hecho, ésa fue la causa que motivó el desarrollo de las técnicas de clustering en subespacios.

Algo más formalmente: cuando aumenta la dimensionalidad de los datos, el número de datos de los que disponemos aumenta linealmente pero el número de regiones en el espacio n -dimensional aumenta exponencialmente. Imaginemos que tenemos un fruto como una naranja. En tres dimensiones, la pulpa de la naranja, que corresponde al 90 % del radio de la naranja, supone unas tres cuartas partes de su volumen, por lo que sólo desperdiciamos una fracción de la naranja cuando nos la comemos.

Si aumentamos a diez dimensiones, manteniendo el 90 % del radio de la naranja para la pulpa, el volumen de la pulpa será inferior al 35 % del volumen total de nuestra hiper-naranja. Al llegar a 100 dimensiones, la pulpa sólo ocupará 26 millonésimas partes del volumen total de la naranja, que esencialmente será sólo cáscara y nunca terminaremos de pelarla.

Como recuerda Pedro Domingos, en los mapas medievales, las zonas no exploradas se marcaban con dragones y otras criaturas mitológicas (o con la frase *hic sunt dracones*, aquí hay dragones). En un espacio multidimensional, muy habitual en problemas de minería de datos, los ejemplos repartidos por él están más cerca de una cara del hipercubo que lo define (la cáscara de la naranja) que de su vecino más cercano. Los dragones, por tanto, están por todas partes y llaman a su puerta.

En aprendizaje automático, conforme aumenta la dimensionalidad de los datos, no sólo aumenta el coste computacional necesario para resolver el problema. El problema en sí se hace más difícil, incluso intratable. En numerosas ocasiones, por suerte para nosotros, no todas las dimensiones son realmente relevantes y podremos reducir su número para que nuestras técnicas de aprendizaje puedan obtener mejores resultados. Matemáticamente, nuestro objetivo es reducir el espacio original a una variedad [*manifold*] en la que el aprendizaje resulte viable. Una variedad, o *manifold* en inglés, es un objeto geométrico que generaliza la noción intuitiva de curva (1-variedad) y de superficie (2-variedad) a cualquier número de dimensiones. Es lo que nos permite utilizar latitud y longitud en las coordenadas GPS, dar una dirección especificando calle y número o, simplemente, señalar con el dedo la dirección hacia la que debe dirigirse un turista despistado. En ninguna de esas situaciones recurrimos a las coordenadas tridimensionales de nuestro destino. Nos basta con dos para orientarnos.

¿Qué técnicas podemos utilizar para reducir el número de dimensiones en las que vienen expresados nuestros datos? Básicamente, dos diferentes: seleccionar un subconjunto de dimensiones de las que ya dis-

El volumen de una hiperesfera en un espacio n -dimensional viene dado por la expresión $\pi^{n/2}r^n/\Gamma(n/2 + 1)$.

ponemos (selección de características) o construir un nuevo marco de referencia que utilice dimensiones diferentes a las originales (extracción de características).

Selección de características

La solución más sencilla a un problema derivado de la alta dimensionalidad de los datos es, simplemente, descartar dimensiones hasta que el problema sea lo suficientemente sencillo como para poder resolverlo con técnicas tradicionales de aprendizaje. No es lo más elegante pero, en ocasiones, funciona relativamente bien.

Obviamente, si descartamos características de las que disponemos en nuestros datos, estamos desperdiciando información que previamente nos hemos molestado en recopilar. Viene a ser como eliminar ejemplos de nuestro conjunto de datos porque no disponemos de técnicas lo suficientemente escalables para procesar todos los datos disponibles (algo que algunos llaman selección de instancias para quedar bien), salvo que ahora lo que seleccionamos son columnas de nuestro conjunto de datos en vez de filas del mismo.

En vez de eliminar características a ojo de buen cubero, lo suyo es que insertemos el proceso de selección de características en el propio proceso de aprendizaje. ¿Cómo se integra la selección de características en ese proceso de aprendizaje?

- *Como un filtro previo [filter]*

Un algoritmo de selección de características selecciona las variables más interesantes de nuestro conjunto de datos como paso previo a la construcción de un modelo de aprendizaje. La selección se realiza como una etapa de preprocesamiento de los datos, de forma independiente a la construcción del modelo. El criterio utilizado para seleccionar una variable puede ser, por ejemplo, la correlación de la variable con el atributo que deseamos predecir.

Las selección de características mediante filtros es eficiente computacionalmente y suele ser robusta frente al sobreaprendizaje, aunque tiende a seleccionar conjuntos de variables redundantes. Además, los resultados obtenidos pueden variar dependiendo del conjunto de entrenamiento disponible: al realizarse como una etapa de preprocesamiento, no tenemos garantías de que la selección realmente contribuya a mejorar el rendimiento del modelo que se construye posteriormente.

- *Envolviendo al método de aprendizaje [wrapper]*

Se van seleccionando conjuntos de variables con los que se van construyendo modelos alternativos. La evaluación de esos modelos nos sirve para ir decidiendo qué variables deberían incluirse en el modelo final.

En realidad, la diferencia esencial entre el *big data* y la estadística de toda la vida es, precisamente, que en *big data* no se muestran los datos. Se utilizan todos.

Dado que el número de combinaciones crece exponencialmente con el número de variables, lo habitual es utilizar un algoritmo *greedy* que, en cada iteración, decide qué variable añadir como entrada para la construcción de nuestro modelo.

El número de variables consideradas va aumentando mientras eso ayude a mejorar el rendimiento de nuestro modelo. Cuando dejan de mejorar los resultados observados, se deja de añadir nuevas variables y se devuelve el mejor modelo encontrado, que usará sólo un subconjunto del conjunto original de variables. El rendimiento del modelo derivado de un subconjunto de características puede evaluarse utilizando un conjunto de validación separado de los conjuntos de entrenamiento y de prueba, para no sesgar la estimación de la calidad del modelo final resultante, o mediante un proceso interno de validación cruzada.

A diferencia de las estrategias de preprocessamiento que seleccionan características a priori, el método *wrapper* permite aprovechar posibles interacciones entre distintas variables. Por el lado negativo, su coste computacional es mucho más elevado (hay que construir múltiples modelos utilizando distintos subconjuntos de variables) y se corre el riesgo de sobreajustar los datos de entrenamiento (sobreaprendizaje, *overfitting*).

- *Incrustada en el propio algoritmo de aprendizaje [embedded]*

Algunas técnicas de aprendizaje llevan ya incorporadas heurísticas que, indirectamente, les permiten seleccionar características. Por ejemplo, el propio proceso de selección de atributos por los que ramificar un árbol de decisión hace que los algoritmos de construcción de árboles de decisión seleccionen automáticamente el conjunto de variables más adecuado para diferenciar unas clases de otras. Las máquinas de vectores de soporte, en cierto modo, también realizan una selección de variables, aunque en este caso sería más correcto hablar de extracción de características...

Extracción de características

La alternativa a seleccionar un subconjunto de las características ya disponibles es crear un conjunto nuevo de características en el que nos resulte más sencillo trabajar con los datos. Ese proceso de extracción o transformación de características nos ayudará a reducir la dimensionalidad de los datos sólo si existe cierta redundancia en ellos, lo que suele ser habitual.

Para analizar la redundancia en los datos, por ejemplo, podemos medir la correlación existente entre parejas de variables. Calculando la matriz de covarianza de los datos, podemos realizar un proceso de análisis de componentes principales, una de las técnicas estadísticas

más antiguas, propuesta por Karl Pearson en 1901.⁸³ El análisis de componentes principales [PCA: *Principal Component Analysis*] es una de las herramientas clave del análisis estadístico de datos. Para algunos con inclinación matemática, el análisis de componentes principales es para el aprendizaje no supervisado como la regresión lineal para el aprendizaje supervisado.

Intuitivamente, el análisis de componentes principales lo que hace es rotar los ejes sobre los que se definen las dimensiones con las que describimos nuestros datos. Los ejes se seleccionan de tal forma que se minimiza la varianza de los residuos (la dispersión de los puntos correspondientes a los datos con respecto a los ejes definidos por los componentes principales). De esa forma, nuestros datos originales se proyectan en subespacios en los que su varianza se maximiza. Normalmente, quedándonos sólo con los primeros componentes principales, tendremos suficiente para representar la variabilidad que existe en nuestros datos, lo que nos permitirá reducir su dimensionalidad. En la jerga propia del área, los primeros componentes principales explican la mayor parte de la varianza de los datos.

Para calcular los componentes principales utilizados en PCA, primero se calcula la matriz de covarianza Σ de los datos. Si tenemos d dimensiones, la matriz será de tamaño $d \times d$:

$$\Sigma = X X^\top$$

Una vez que tenemos esa matriz, obtenemos los k mayores eigenvectores, autovectores o vectores propios de la matriz de covarianza, con $k \ll d$, algo que podemos hacer utilizando un algoritmo numérico de tipo iterativo (o con la ayuda de una biblioteca de cálculo numérico que ya proporcione esa funcionalidad). Esos eigenvectores serán nuestros componentes principales. Dado que la matriz de covarianza es simétrica, es diagonalizable y sus eigenvectores pueden normalizarse para que sean ortonormales:

$$\Sigma = X X^\top = W D W^\top$$

Sin embargo, el cálculo de los eigenvectores W de la matriz de covarianza puede ser demasiado costoso o puede que incluso nos enfrentemos a un problema con una dimensionalidad tan elevada que ni siquiera podamos calcular la matriz de covarianza (hay problemas en los que podemos tener decenas de miles o, incluso, millones de características diferentes). En esos casos, podemos recurrir a otras técnicas matemáticas, como es el caso de la descomposición en valores singulares [SVD: *Singular Value Decomposition*].

Dada una matriz con los datos de los que disponemos X , de tamaño $n \times d$, podemos descomponer esa matriz como

$$X = U D V^\top$$

⁸³ Karl Pearson F.R.S. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine Series 6*, 2(11):559–572, 1901. DOI: 10.1080/14786440109462720

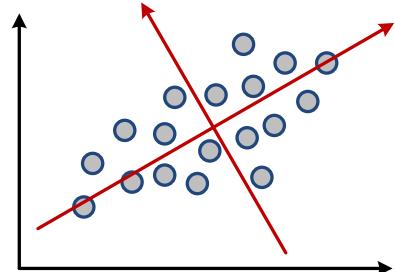


Figura 21: Análisis de componentes principales (PCA): Los ejes inclinados corresponden a los dos primeros componentes principales del conjunto de datos mostrado en la imagen.

donde U es una matriz ortogonal del mismo tamaño que nuestra matriz de datos ($n \times d$), D es una matriz diagonal ($d \times d$) y V es otra matriz ortogonal ($d \times d$). Los eigenvectores de la matriz de covarianza Σ serán las columnas de la matriz U , dado que $\Sigma = XX^\top$, $X = UDV^\top$ y $V^\top V = I$, de donde se deduce que $\Sigma = UD^2U^\top$.

Para reducir la dimensionalidad de los datos, basta con que nos quedemos únicamente con los k primeros componentes principales, usando $k \ll d$:

$$\tilde{X} \approx \tilde{U}\tilde{D}\tilde{V}^\top$$

donde estamos aproximando nuestra matriz de datos X ($n \times d$) como el producto de tres matrices más pequeñas: \tilde{U} ($n \times k$), \tilde{D} ($k \times k$) y \tilde{V} ($k \times d$).

La descomposición en valores singulares, SVD, nos permite proyectar nuestros datos en un espacio de menos dimensiones. Técnicas de este tipo, por ejemplo, nos permiten construir sistemas de reconocimiento facial. Imaginemos que partimos de una pequeña imagen de 64×64 píxeles. Nuestros datos de entrada contienen, pese a lo reducido de las imágenes, un total de 4096 dimensiones. Sin embargo, analizando esas imágenes podemos darnos cuenta de que, tal vez, los 50 primeros eigenvectores describen el 90 % de la varianza observada en los datos. Con quedarnos sólo con las 50 dimensiones dadas por esos 50 primeros eigenvectores podemos reconstruir los datos sin perder demasiada calidad. Además, podemos emplear esas 50 características extraídas para construir un clasificador que sirva para reconocer a personas concretas. La extracción de características nos ha permitido reducir espectacularmente la dimensionalidad de nuestro problema: de 4096 dimensiones inicialmente a sólo 50 dimensiones, lo que lo hace mucho más tratable.

También existen técnicas no lineales de reducción de la dimensionalidad de los datos, p.ej. Isomap.⁸⁴ Isomap conecta cada punto en un espacio multidimensional (una cara) con todos sus puntos cercanos (caras muy similares), calcula las distancias más cortas entre todos los pares de puntos en la red resultante y encuentra un conjunto reducido de coordenadas que aproxima esas distancias de la mejor forma posible. Podemos pensar que Isomap lo que hace es indicarnos el punto kilométrico en una carretera que rodea una bahía, algo que tiene mucho más sentido que utilizar las coordenadas geográficas de cada punto si luego queremos establecer similitudes entre puntos cercanos. Para llegar de un punto a otro, salvo que estemos dispuestos a nadar, tendremos que bordear la bahía siguiendo la carretera. La distancia entre las coordenadas geográficas de origen y destino no resulta indicativa del tiempo que tardaremos en llegar de un punto a otro, mientras que la distancia medida sobre las coordenadas obtenidas por Isomap sí lo es.

La identificación de componentes principales, lineales o no, no es la única técnica de extracción de características que podemos utilizar. Por

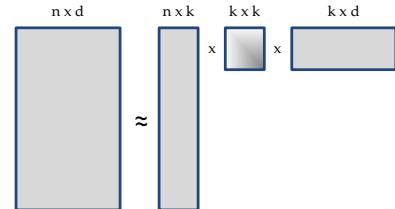


Figura 22: Descomposición en valores singulares (SVD): Aproximación de la matriz de datos como un producto de tres matrices más pequeñas.

⁸⁴ Joshua B. Tenenbaum, Vin de Silva, y John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500): 2319–2323, 2000. ISSN 0036-8075. DOI: 10.1126/science.290.5500.2319

ejemplo, en el análisis de documentos realizado en minería de textos [*text mining*], es relativamente común el uso de técnicas de modelado de temas [*topic modeling*] como LDA [*Latent Dirichlet Allocation*], una técnica propuesta en 2003 por David Blei, Andrew Ng y Michael Jordan.⁸⁵ LDA es un modelo bayesiano en el que cada documento de una colección se modela como una mezcla de un conjunto de temas: la aparición de cada palabra del documento es atribuible a alguno de los temas sobre los que versa el documento. Es decir, un documento se describe, en vez de por las palabras que en él aparecen, por los temas que trata. Una forma más de reducir la dimensionalidad de los datos.

Ingeniería de características

En muchos problemas, demasiados tal vez, el secreto de encontrar una solución adecuada utilizando técnicas de aprendizaje automático reside en saber confeccionar el conjunto de características más adecuado para el problema particular que tengamos entre manos. Este proceso, que puede resultar arduo y muy laborioso, es el que tradicionalmente se ha utilizado para resolver muchos problemas. En ocasiones recibe el pomposo nombre de “ingeniería de características” [*feature engineering*].

La ingeniería de características, de forma implícita, es la idea que subyace tras la búsqueda de características particulares que puedan resultar clave para resolver problemas concretos como la predicción de enlaces en redes complejas⁸⁶ o la identificación de objetos en visión artificial.^{87,88} Al menos, lo era hasta la llegada de AlexNet, la red neuronal mediante la cual un par de estudiantes de doctorado consiguieron superar a todas las técnicas tradicionales que se habían venido desarrollando manualmente durante décadas, desatando la revolución del *deep learning* en visión artificial.⁸⁹

La ingeniería de características es un proceso que consiste, esencialmente, en identificar conexiones entre variables y empaquetarlas en una nueva variable con la que será más sencillo resolver el problema original. Diseñar características a mano resulta lento, incómodo y extremadamente costoso. Requiere un conocimiento experto acerca del problema particular y, además, los resultados obtenidos quedan limitados a un único dominio de aplicación, el del problema para el que se diseñan las características. Un proceso demasiado *ad hoc*.

Las técnicas de *deep learning* también realizan ingeniería de características, pero de forma completamente automática. Ahí reside su principal atractivo. Las redes neuronales son capaces de determinar qué características son realmente relevantes y, por tanto, cuáles son las que deberían utilizarse en cada momento para resolver un problema particular. Algo de lo que son incapaces otras técnicas de aprendizaje automático, a las que les debemos proporcionar los datos ya preparados.

⁸⁵ David M. Blei, Andrew Y. Ng, y Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, March 2003. ISSN 1532-4435. URL <http://jmlr.csail.mit.edu/papers/v3/blei03a.html>

Esta técnica fue propuesta independientemente por Jonathan Pritchard, Matthew Stephens y Peter Donnelly para el estudio genético de poblaciones tres años antes.

Jonathan K. Pritchard, Matthew Stephens, y Peter Donnelly. Inference of population structure using multilocus genotype data. *Genetics*, 155(2):945–959, 2000. ISSN 0016-6731. URL <http://www.genetics.org/content/155/2/945>

⁸⁶ Víctor Martínez, Fernando Berzal, y Juan Carlos Cubero. Adaptive degree penalization for link prediction. *Journal of Computational Science*, 13:1–9, 2016. DOI: 10.1016/j.jocs.2015.12.003

⁸⁷ David G. Lowe. Object recognition from local scale-invariant features. En *Proceedings of the 7th IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157, 1999. DOI: 10.1109/ICCV.1999.790410

⁸⁸ Jie Chen, Shiguan Shan, Chu He, Guoying Zhao, Matti Pietikainen, Xilin Chen, y Wen Gao. WLD: A Robust Local Image Descriptor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1705–1720, Sept 2010. ISSN 0162-8828. DOI: 10.1109/TPAMI.2009.155

⁸⁹ Alex Krizhevsky, Ilya Sutskever, y Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. En *NIPS’2012 Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL <https://goo.gl/Ddt8Jb>

Deep Learning

Las redes neuronales, conocidas actualmente bajo el término paraguas de deep learning, proporcionan uno de los mecanismos mediante los que se puede conseguir que un ordenador aprenda. Los conexionistas se inspiran en el cerebro humano y construyen modelos informáticos formados por múltiples unidades relativamente simples a las que denominan neuronas. Estas neuronas, o elementos de procesamiento, se conectan entre sí para formar redes neuronales artificiales. Mediante la manipulación de las conexiones entre las neuronas de la red se consigue que la red neuronal tenga el comportamiento deseado.

Desde el punto de vista abstracto, las redes neuronales representan conceptos mediante patrones de actividad en una red distribuida de neuronas. A diferencia de los modelos simbólicos, que representan de forma explícita el conocimiento mediante reglas lógicas, las redes neuronales representan el conocimiento de forma implícita, a través de los pesos con los que se modelan las conexiones entre neuronas. El procesamiento subsimbólico de datos típico de las redes neuronales, aunque las hace menos interpretables que los modelos simbólicos, dota a las redes de una capacidad automática de generalización, muy útil a la hora de resolver problemas complejos de aprendizaje.

Cuando se entrena una red neuronal, el ajuste de sus pesos para aprender un nuevo concepto afecta a la representación distribuida de otros conceptos con patrones de actividad similares. Cuando se utiliza la red neuronal, ante patrones de actividad similares pero no iguales a los vistos por ella durante su fase de entrenamiento, la red es capaz de generalizar con éxito donde otras técnicas de aprendizaje automático fracasan estrepitosamente.

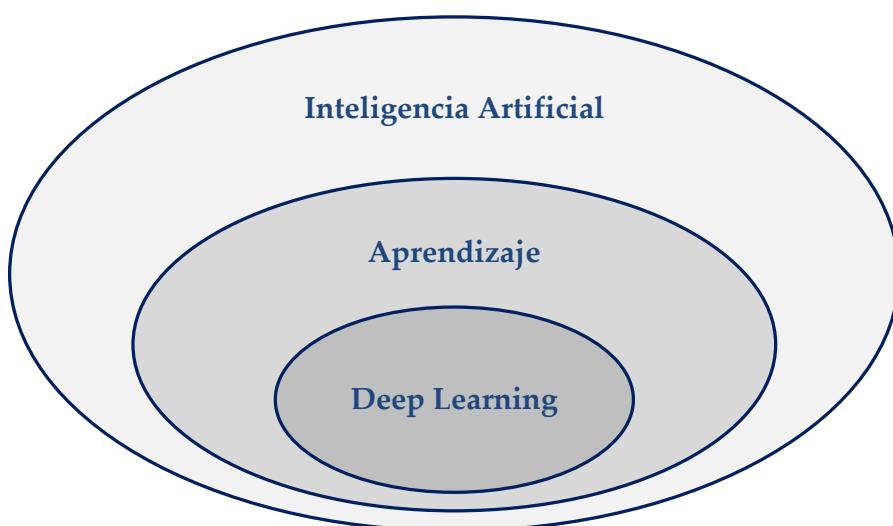


Figura 23: El *deep learning* como un subconjunto de técnicas de aprendizaje automático, que a su vez es una de las disciplinas englobadas dentro del campo de la Informática conocido como Inteligencia Artificial.

Las técnicas de *deep learning* son un subconjunto dentro de un subconjunto. Forman parte del conjunto de técnicas de aprendizaje automático, que a su vez son un subconjunto de las técnicas utilizadas en Inteligencia Artificial.

- La Inteligencia Artificial abarca una amplia gama de técnicas y herramientas utilizadas para que ordenadores y robots muestren un comportamiento similar al comportamiento inteligente de un ser vivo. Uno de los rasgos de ese comportamiento es, precisamente, su capacidad de aprender.
- El aprendizaje automático cubre una amalgama de estrategias diferentes que se han propuesto para que un ordenador sea capaz de mejorar con la experiencia su rendimiento al resolver problemas concretos. Simbólicos, analógicos, bayesianos, evolutivos y conexionistas promueven diferentes formas de lograr un objetivo común.
- Los conexionistas son los especialistas en aprendizaje automático que desarrollan técnicas de *deep learning*. Mediante el uso de herramientas matemáticas de optimización, ajustan la configuración de una red neuronal artificial para conseguir realizar tareas que, hasta hace poco, estaban vedadas a los ordenadores.

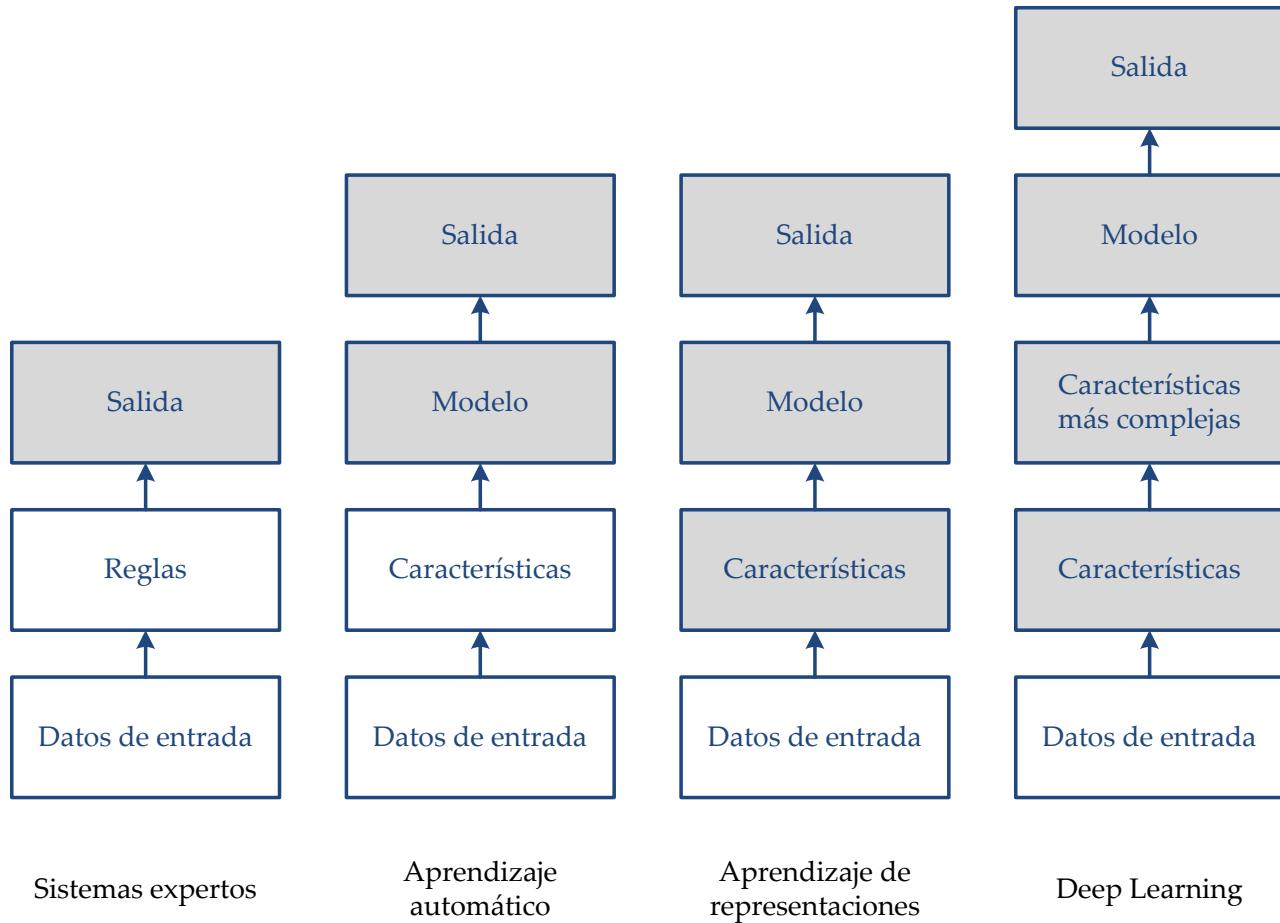
En pocas décadas, las técnicas de Inteligencia Artificial han evolucionado de forma espectacular.

Los primeros éxitos comerciales de la I.A. fueron sistemas expertos que emulaban el razonamiento de un experto en un dominio de aplicación específico aprovechando el conocimiento de expertos humanos, que se codificaba manualmente en forma de reglas.

Posteriormente, las técnicas de aprendizaje automático y de minería de datos consiguieron resolver problemas construyendo modelos a partir de conjuntos de datos, en los que el experto seleccionaba y diseñaba manualmente las características, atributos o variables que resultasen más útiles desde su punto de vista. En ocasiones, con ayuda de técnicas de selección y extracción de características.

A continuación, comenzaron a aparecer técnicas que eran capaces de descubrir automáticamente, a partir de los datos, las características que pudiesen ser útiles en la resolución de un problema. Esas técnicas recibieron el apelativo de técnicas de aprendizaje de representaciones [*representation learning*].

La continuación natural de esa progresión, en la que cada vez se automatizan más etapas del proceso de aprendizaje, nos lleva hasta las técnicas de *deep learning*. Las técnicas de *deep learning* son capaces de descubrir características a partir de los datos, como las técnicas de aprendizaje de representaciones, pero también son capaces de crear nuevas características a partir de otras características. De esa forma, son capaces



no sólo de identificar características que nos ayuden a discriminar mejor en función de la situación en la que nos encontramos, sino de aumentar el nivel de abstracción al que se trabaja mediante la creación de jerarquías con varios niveles de características, en las que nuevas características se descubren de forma automática a partir de las características del nivel anterior.

En otras palabras, las técnicas de *deep learning* realizan ingeniería de características de forma completamente automática. Las redes neuronales que se emplean están formadas por múltiples capas de neuronas, como el córtex cerebral, el cerebelo o la retina humana. Cada capa permite construir nuevas características a partir de las características identificadas por las capas anteriores, lo que resulta especialmente útil en el análisis de datos no estructurados como señales de audio o imágenes. Es la capacidad de formar modelos a distintos niveles de abstracción la que ha permitido construir sistemas automáticos de reconocimiento de voz o de identificación de objetos en imágenes con un rendimiento similar al

Figura 24: Evolución de las técnicas de Inteligencia Artificial, desde los sistemas expertos basados en reglas hasta el *deep learning*. Las zonas sombreadas corresponden a las etapas del proceso que se automatizan.

conseguido por el ser humano.

La ingeniería automática de características resulta interesante porque la alternativa es construir esas características manualmente, en un proceso arduo, propenso a errores y a medida para cada problema concreto. El *deep learning* automatiza ese proceso manual. Eso sí, para que funcione correctamente necesita disponer de datos. Muchos datos.

Afortunadamente, hoy en día es relativamente fácil recopilar grandes cantidades de datos y la potencia de cálculo de los ordenadores actuales hace factible el procesamiento de grandes volúmenes de datos en un tiempo razonable.

Alguno se estará preguntando, ¿pero eso no lo hacían ya las redes neuronales artificiales desde los años 80 del siglo pasado? En realidad, sí que lo hacían. Las ideas que subyacen a las modernas técnicas de *deep learning* son, esencialmente, las mismas con las que se entrenaban las redes neuronales hace décadas.⁹⁰ El salto cualitativo se ha dado gracias al refinamiento de los algoritmos de entrenamiento de redes neuronales, que ahora pueden tener muchas más capas que antes, así como a la disponibilidad de conjuntos de datos enormes y de la potencia de cálculo necesaria para poder ajustar el número inmenso de parámetros de una red neuronal artificial con múltiples capas y un número elevado de neuronas. Lejos aún del número de neuronas del cerebro humano, del orden de 10^{11} , pero lo suficientemente elevado como para resolver problemas concretos de forma efectiva.

¿Cuándo se produjo ese salto cualitativo? En el mundo académico, el salto se inició en 2006 con la publicación de tres trabajos en los que se proponían algoritmos para entrenar redes neuronales más complejas que las que hasta entonces se habían podido utilizar. A los autores de esos tres trabajos ya los hemos presentado antes: Geoffrey Hinton,⁹¹ Yoshua Bengio,⁹² y Yann LeCun.⁹³ En esos momentos, se encontraban en las universidades de Toronto, Montréal y Nueva York, respectivamente. Eran de los pocos que habían seguido trabajando en redes neuronales artificiales después de que el interés en ellas decayese y otras técnicas, como las máquinas de vectores de soporte, acaparasen casi toda la atención de los expertos en aprendizaje automático.

Poco tiempo después, algoritmos derivados de esos primeros resultados publicados en 2006 se comenzaron a emplear en la resolución de una amplia variedad de problemas de aprendizaje. Problemas que, hasta ese momento, se habían mostrado casi intratables para las técnicas de aprendizaje automático. Las redes neuronales artificiales están disfrutando ahora de una nueva época dorada gracias a que resultan útiles por su capacidad para resolver problemas complejos cuya solución había eludido durante décadas los esfuerzos de innumerables investigadores. Basten dos ejemplos notables:

⁹⁰ Geoffrey E. Hinton, James L. McClelland, y David E. Rumelhart. Distributed Representations. En David E. Rumelhart, James L. McClelland, y PDP Research Group, editores, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, pages 77–109. MIT Press, 1986. ISBN 026268053X

⁹¹ Geoffrey E. Hinton, Simon Osindero, y Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006. DOI: 10.1162/neco.2006.18.7.1527

⁹² Yoshua Bengio, Pascal Lamblin, Dan Popovici, y Hugo Larochelle. Greedy layer-wise training of deep networks. En *NIPS'2006 Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 153–160, 2006a. URL <https://goo.gl/KyBc9x>

⁹³ Marc'Aurelio Ranzato, Christopher S. Poultney, Sumit Chopra, y Yann LeCun. Efficient learning of sparse representations with an energy-based model. En *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 1137–1144, 2006. URL <https://goo.gl/pKBzV0>

- El uso de redes neuronales en sistemas de reconocimiento de voz se generalizó tras los trabajos de investigadores de Microsoft Research, que redujeron notablemente la tasa de error de ese tipo de sistemas incorporando redes neuronales a los mismos.⁹⁴ Hasta ese momento, los sistemas de reconocimiento de voz no resultaban comercialmente viables, salvo en entornos muy limitados (con vocabulario reducido o en entornos controlados, sin demasiado ruido de fondo). Una reducción cuantitativa de la tasa de error posibilitó un salto cualitativo en los sistemas de reconocimiento de voz.
- La victoria de dos doctorandos de Hinton, Alex Krizhevsky e Ilya Sutskever, en una conocida competición de reconocimiento de objetos en imágenes consiguió un efecto similar en el campo de la visión artificial.⁹⁵ Desde entonces, las redes neuronales han sido capaces de llegar a ser competitivas con el ser humano a la hora de identificar objetos en imágenes. Ninguna otra técnica, manual o automática, les hace sombra hoy en día.

Los conexiónistas clásicos se han reciclado ahora como expertos en *deep learning*. Yoshua Bengio ha publicado un libro de texto excelente en el que se describen muchas de las técnicas de *deep learning*. El libro, disponible en <http://www.deeplearningbook.org/>, lo ha escrito Bengio en colaboración con Aaron Courville, también profesor de la Universidad de Montréal, e Ian Goodfellow, investigador en OpenAI, una organización sin ánimo de lucro que pretende desarrollar una inteligencia artificial general de forma segura.⁹⁶ Muchas de las técnicas que se describen en ese libro aparecían ya, aunque fuese de forma algo más primitiva, en multitud de libros de texto^{97,98} que se publicaron en los años 90, como resultado de la anterior época dorada de la que disfrutaron las redes neuronales artificiales. Por desgracia, sólo uno de ellos se tradujo entonces al español.⁹⁹

Tras múltiples altibajos a lo largo de las últimas décadas, estamos nuevamente en una era dorada para las redes neuronales artificiales, ahora bajo la denominación de *deep learning*. ¿Será ésta la definitiva?

Características clave del deep learning

Las redes neuronales artificiales se inspiran en lo (relativamente poco) que se sabe acerca del funcionamiento del cerebro humano. No obstante, las redes neuronales artificiales no siempre se ajustan de forma estricta al comportamiento de un cerebro biológico. De hecho, muchas redes neuronales artificiales de las que se utilizan para resolver problemas de interés práctico guardan un parecido más bien escaso con sus homólogas biológicas.

Lo realmente interesante de las redes neuronales artificiales es que

⁹⁴ George E. Dahl, Dong Yu, Li Deng, y Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Trans. Audio, Speech & Language Processing*, 20(1):30–42, 2012. DOI: 10.1109/TASL.2011.2134090

⁹⁵ Alex Krizhevsky, Ilya Sutskever, y Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. En *NIPS'2012 Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL <https://goo.gl/Ddt8Jb>

⁹⁶ Ian Goodfellow, Yoshua Bengio, y Aaron Courville. *Deep Learning*. MIT Press, 2016. ISBN 0262035618. URL <http://www.deeplearningbook.org>

⁹⁷ Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1996. ISBN 0198538642

⁹⁸ Fredric M. Ham y Ivica Kostanic. *Principles of Neurocomputing for Science and Engineering*. McGraw-Hill, 2000. ISBN 0070259666

⁹⁹ James A. Freeman y David M. Skapura. *Redes neuronales: Algoritmos, aplicaciones y técnicas de programación*. Addison-Wesley / Díaz de Santos, 1993. ISBN 020160115X

son capaces de resolver problemas que ningún programador humano había sido capaz de resolver con la ayuda de un ordenador. En realidad, porque no podría. Las redes neuronales artificiales están especialmente indicadas para problemas complejos en los que existen multitud de casos particulares. Esos casos particulares, que un programador debería ir considerando uno a uno para integrarlos en un algoritmo convencional, son los que imposibilitan una solución codificada manualmente.

Casi todos los algoritmos de *deep learning* se pueden describir como instancias particulares de una receta bastante simple, que aplicamos para resolver un problema concreto de aprendizaje automático:

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

No hay más. En realidad, es la misma estrategia que se utiliza en otras muchas técnicas de aprendizaje automático: datos, función, modelo y optimización. Un esquema general que nos permite ver a todas las técnicas de aprendizaje desde un prisma común, más que como una lista interminable de algoritmos, cada uno con sus diferentes motivaciones (y justificaciones más o menos cogidas con alfileres, ya que nunca dejan de ser técnicas heurísticas).

¿Qué es lo que diferencia al *deep learning* de otras familias de técnicas de aprendizaje? Esencialmente, su capacidad de abstracción. Su capacidad de formar jerarquías de conceptos utilizando múltiples niveles de representación.

La abstracción, desde el punto de vista psicológico, es un proceso mediante el cual se identifican los componentes esenciales de algo con el objetivo de conservar en la abstracción sus rasgos más relevantes y prescindir de los que no lo son. Consiste en separar la esencia y los accidentes de lo que se esté analizando. En definitiva, lo mismo que hacemos cuando construimos un modelo (de aprendizaje automático o de cualquier otro tipo). En Informática, se emplea constantemente para aislar un componente de su entorno y ocultar detalles de implementación, lo que nos permite resolver problemas cada vez más complejos. No en vano, según Grady Booch, un conocido ingeniero de IBM, la historia completa del desarrollo de software es una historia de crecientes niveles de abstracción. La abstracción es la forma en que los humanos tratamos con la complejidad del mundo que nos rodea.

Las técnicas de *deep learning* pretenden automatizar ese proceso de abstracción, creando nuevas abstracciones sobre otras ya existentes. La topología más habitual de las redes neuronales está formada por múltiples capas, cada una de las cuales recibe como entrada la salida de la capa anterior. De esa forma, se consigue un sistema modular en el que cada capa proporciona un mayor nivel de abstracción con respecto a la entrada que recibe la red neuronal.

Por ejemplo, cuando se trabaja con imágenes, la entrada de la red será una matriz de píxeles, cada uno de ellos con un nivel de intensidad asociado. En el caso de las imágenes en color, la red recibirá un conjunto de matrices, una por cada canal de la imagen (rojo, verde, azul). En una primera capa, la red será capaz de identificar dónde aparecen bordes en la imagen, fronteras entre zonas con diferente nivel de intensidad en sus píxeles. Capas posteriores serán capaces de combinar fronteras para formar esquinas, esquinas para formar facetas, facetas para formar objetos, objetos para componer escenas... Y así hasta que la red sea capaz de identificar la presencia de un objeto concreto. Independientemente de si aparece más o menos iluminado en la imagen; ligeramente desplazado en una dirección u otra; en vertical, en horizontal o sutilmente inclinado hacia un lado; más o menos cerca de la cámara que captó la imagen; parcialmente oculto por la presencia de otros objetos en la escena... Un sinfín de posibilidades para las que resultaría imposible diseñar un algoritmo secuencial robusto que fuese capaz de tenerlas todas en cuenta.

Los niveles de abstracción de una red neuronal multicapa le permiten seleccionar en cada momento las características comunes de los ejemplos que ha de reconocer, aquéllos para lo que ha sido diseñada expresamente. Simultáneamente, la red es capaz de ignorar los pequeños cambios locales propios de cada ejemplo particular, quedándose, de forma efectiva, con la esencia de la clase a la que pertenece el ejemplo.

Las técnicas de *deep learning* parten de la idea base de que, si somos capaces de aprender con éxito múltiples niveles de representación, podremos generalizar correctamente. En lugar de depender de un algoritmo que se limite a extraer características concretas de los datos disponibles, el *deep learning* proporciona herramientas que nos permiten construir características a partir de otras características. Repitiendo el proceso, podemos construir modelos jerárquicos de características. Esos niveles de características proporcionan múltiples niveles de representación, que corresponden a múltiples niveles de abstracción. Esos niveles de abstracción nos permiten olvidarnos de detalles irrelevantes y desentrañar los factores clave que explican aquello que estemos estudiando. Exactamente lo mismo que hace la ciencia para explicar los fenómenos naturales.

De acuerdo, entonces, en que la detección de características y la construcción de nuevos niveles de características a partir de los anteriores pueden ayudarnos a resolver problemas complejos. Pero... ¿qué sucede

con la maldición de la dimensionalidad? ¿No estamos intentando resolver un problema difícil recurriendo a un problema combinatorio que resulta intratable?

La clave para soslayar el problema de la maldición de la dimensionalidad nos la proporciona una característica clave de las redes neuronales artificiales: su “composicionalidad”. Podemos ver las redes neuronales como si fuesen conjuntos de bloques de LEGO. Cada bloque tiene sus características propias e, individualmente, está algo limitado. Sin embargo, los bloques se diseñan para que se puedan interconectar entre sí, lo que permite construir modelos completos que sí resultan adecuados para resolver problemas complejos.

Yoshua Bengio distingue dos tipos de composición en las redes neuronales artificiales:¹⁰⁰

- *Composición paralela: Representaciones distribuidas*

Las redes neuronales representan de forma distribuida los conceptos, mediante las conexiones que se forman entre unas neuronas y otras, denominadas sinapsis. En las redes neuronales artificiales, la fuerza de las sinapsis se representa numéricamente mediante un peso. Dicho peso tendrá un valor positivo cuando la sinapsis sea excitatoria; esto es, cuando la activación de la neurona presináptica facilite la activación de la neurona postsináptica. El peso será negativo en las sinapsis inhibitorias, cuando la activación presináptica inhibe la activación postsináptica.

La representación distribuida está en el origen de la I.A. conexionista, que en los años 80 surgió con fuerza como alternativa a la I.A. simbólica, más tradicional. De hecho, uno de los grupos de investigación más conocidos de aquella época era el dirigido por el psicólogo James ‘Jay’ McClelland en la Universidad de Stanford: el grupo PDP [*Parallel Distributed Processing*]. Los conexionistas como McClelland interpretan las funciones cognitivas (aprendizaje, memoria, lenguaje y desarrollo cognitivo) como el resultado emergente de la actividad distribuida y paralela de poblaciones de neuronas. El aprendizaje sucede mediante la adaptación de las conexiones entre las neuronas.^{101,102}

Los modelos computacionales basados en esa idea dieron lugar a las redes neuronales artificiales tal como las conocemos hoy. Otras técnicas de aprendizaje, como los árboles de decisión, los clasificadores basados en los vecinos más cercanos o las máquinas de vectores de soporte, necesitan ajustar parámetros locales para cada región del espacio en la que deben operar. Su complejidad está asociada al número de regiones que son capaces de diferenciar. Sin embargo, la representación distribuida propia de las redes neuronales les permite representar un número exponencial de regiones utilizando sólo un número lineal de parámetros. Esos parámetros corresponden a características in-

¹⁰⁰ Yoshua Bengio. Deep Learning: Theoretical Motivations. *Deep Learning Summer School, Montreal, 2015*. URL <https://goo.gl/pFXok6>

¹⁰¹ David E. Rumelhart, James L. McClelland, y the PDP Research Group, editores. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition - Volume 1: Foundations*. MIT Press, 1986b. ISBN 0262181207

¹⁰² David E. Rumelhart, James L. McClelland, y the PDP Research Group, editores. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition - Volume 2: Psychological and Biological Models*. MIT Press, 1986c. ISBN 0262132184

dividuales con significado, que pueden aprenderse individualmente y, en principio, podrían reutilizarse. No siempre seremos capaces de darles una interpretación semántica, de tipo simbólico, pero son esas representaciones distribuidas las que dotan a las redes neuronales artificiales de su capacidad de generalización no local.

En otros modelos de aprendizaje no paramétricos, como los árboles de decisión o los clasificadores basados en instancias tipo k-NN, no podemos decir nada acerca de un ejemplo que caiga en una zona del espacio de datos de entrada que no estuviese representado en el conjunto de entrenamiento. Las redes neuronales sí serán capaces de hacerlo: son capaces de predecir algo con sentido acerca de lo que no han visto nunca. Según Bengio, la esencia de la generalización.

■ *Composición secuencial: Niveles de abstracción*

Durante el siglo XX, no se hizo demasiado hincapié en el estudio de redes neuronales artificiales profundas. No fue hasta la primera década del siglo XXI cuando se popularizaron los modelos de redes neuronales artificiales con múltiples capas ocultas, denominadas “redes neuronales profundas” [*deep neural networks*], las que dieron lugar al *deep learning*.

¿Por qué no? Porque, desde el punto de vista formal, se había demostrado que una red neuronal multicapa con una única capa oculta servía como aproximador universal. Una red poco profunda [*shallow*], con una única capa de unidades ocultas, era suficiente para representar cualquier función con el grado necesario de exactitud (dados el número suficiente de unidades ocultas, claro está). Esa propiedad matemática de aproximación universal hizo que pocos se planteasen aumentar la complejidad de las redes neuronales haciéndolas más profundas. No resultaba necesario, al menos desde el punto de vista formal.

Ya que hacer las redes más profundas no significa que puedan aprender más cosas (representar más funciones), ¿qué ventaja ofrecen las redes neuronales profundas? Básicamente, la posibilidad de representar la misma función que una red neuronal poco profunda, pero hacerlo de forma más económica, con menos unidades ocultas.

La hipótesis sobre la que se asienta el desarrollo de las redes profundas es que, si la función que estamos intentando aprender tiene características particulares que se pueden obtener mediante la composición de múltiples operaciones, entonces resulta mucho mejor aproximar esa función con una red neuronal profunda. En teoría, el número exponencial de neuronas ocultas que podría necesitar una red neuronal poco profunda podría reducirse a un número polinómico de neuronas si se diseña una red profunda de la topología adecuada.

En el fondo, no es más que otra forma de exponer las tesis de Herbert Simon con respecto al diseño de sistemas complejos.¹⁰³ En la

Se entiende por profunda [*deep*] cualquier red neuronal artificial que tenga más de una capa oculta, aparte de sus capas de entrada y de salida.

¹⁰³ Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, 3rd edition, 1996. ISBN 0262691914

naturaleza, la complejidad a menudo adopta la forma de un sistema jerárquico. Un sistema complejo se compone de subsistemas que, a su vez, tienen sus propios subsistemas. Puede que el origen de esta organización se deba, simplemente, a que un sistema jerárquico puede evolucionar mucho más rápidamente que un sistema no jerárquico de tamaño similar. Las jerarquías no sólo aparecen en el mundo natural, sino que subyacen en muchas de las creaciones del hombre, incluso cuando no tienen conexiones directas con la evolución natural. Al fin y al cabo, montar un reloj mecánico de múltiples piezas es mucho más sencillo si lo vamos haciendo por partes que si pretendemos encajar todas las piezas de golpe. En Ingeniería del Software, las arquitecturas multicapa se repiten por todas partes, desde el sistema operativo hasta los protocolos de comunicación en redes como Internet, desde las aplicaciones de gestión más rutinarias hasta los sistemas más sofisticados de minería de datos. La clave de un buen diseño suele residir en una adecuada selección de los niveles de abstracción que simplifiquen la resolución de un problema complejo.

En el caso del *deep learning*, es la propia topología de la red neuronal la que la hace adecuada para el análisis de sistemas complejos que, en principio, es más que probable que tengan una cierta organización jerárquica interna.

Además, la composición modular de una red neuronal compleja permite que reutilicemos componentes de esa red y compartamos módulos comunes para diferentes aplicaciones. Lo dicho, una red neuronal como un conjunto de piezas de LEGO que podemos combinar de múltiples formas.

Los seres humanos, cuando aprendemos, somos capaces de hacerlo utilizando muy pocas indicaciones. Algo que no sucede con los modelos de aprendizaje automático. Los niños son capaces de aprender nuevas tareas con sólo unos pocos ejemplos, incluso puede que sólo uno. Estadísticamente, sin embargo, es imposible generalizar a partir de un único ejemplo. Una explicación plausible al aprendizaje de un niño es que el niño es capaz de reutilizar conocimiento adquirido en el aprendizaje de otras tareas: el conocimiento previo se utiliza para construir representaciones en las que un único ejemplo puede servir para generalizar adecuadamente. Algo así es lo que podrían llegar a ofrecer las redes neuronales artificiales de composición modular. De hecho, se han propuesto algunas técnicas de aprendizaje semisupervisado y de aprendizaje multitarea [*multi-task learning*] con este objetivo.

Ambos tipos de composición, la paralela asociada a los modelos conexionistas y la secuencial asociada al *deep learning*, permiten que las redes neuronales sean más capaces de generalizar adecuadamente que otras técnicas de aprendizaje automático. La representación distribuida

propia de las redes neuronales les permite generalizar en zonas del espacio de entrada que nunca han visto. La representación modular de las redes profundas, con múltiples niveles de abstracción, les permite extraer características en un contexto que luego pueden aplicar en otro.

¿Hay algo en el *deep learning* que no se pueda conseguir con otras técnicas de aprendizaje automático? Desde el punto de vista teórico, nada. Las técnicas de los años 80, como las redes neuronales multicapa entrenadas con *backpropagation*, los árboles de decisión o las redes bayesianas, ya eran aproximadores universales, por lo que, en principio, son capaces de aprender cualquier cosa que se pueda aprender. De la misma forma, las máquinas de vectores de soporte [*SVMs: Support Vector Machines*] de los años 90 disponen de un algoritmo (relativamente) eficiente para ajustar sus parámetros controlando el sobreaprendizaje. Las SVM, muy populares a finales del siglo XX y comienzos del XXI, proporcionan un algoritmo “inteligente” para seleccionar características y encontrar la mejor forma de resolver un problema de aprendizaje, planteado éste como un problema matemático de optimización.

No obstante, pese a su popularidad, las SVM nunca fueron una buena opción desde el punto de vista de la Inteligencia Artificial, tal como remarcaba Geoffrey Hinton. Esencialmente, porque no dejan de ser una reencarnación de los perceptrones, las redes neuronales artificiales más primitivas. Diseñados a finales de los años 50 por Frank Rosenblatt y popularizados en los años 60 antes de llegar al invierno de la I.A., los perceptrones sólo tienen una capa de neuronas, motivo por el que son incapaces de aprender determinadas cosas. Aunque las máquinas SVM sí son capaces de aprender cualquier cosa, en el fondo se limitan única y exclusivamente a expandir el tamaño de la capa de entrada del perceptrón. Eso sí, son capaces de trabajar implícitamente con un conjunto gigantesco de características, el cual puede ser potencialmente infinito. Sin embargo, carecen de la capacidad de adaptación y de composición de las redes neuronales artificiales utilizadas en *deep learning*: sus características no son adaptativas (viene determinadas por el tipo de kernel utilizado por la máquina SVM) y son incapaces de representar múltiples niveles de abstracción (sólo disponen de una capa de parámetros ajustables, que utilizan para construir características directamente a partir de los vectores de soporte, correspondientes a los datos del conjunto de entrenamiento).

Ahora bien, si, en esencia, las redes neuronales utilizadas en la actualidad no difieren tanto de las que se usaban hace 30 años, ¿por qué se ha popularizado el *deep learning* precisamente ahora? ¿Qué factores han contribuido a revitalizar las redes neuronales artificiales?

Tres son las causas principales: la disponibilidad de conjuntos de datos enormes, la potencia de cálculo proporcionada por las GPU [*Graphical Processing Unit*] y el desarrollo de nuevos algoritmos.

En 1995, dos investigadores de los laboratorios Bell de la AT&T, Larry Jackel y Vladimir Vapnik, se apostaron una cena con Yann LeCun como testigo. Jackel defendía que, en el año 2000, se tendría una explicación formal de por qué las redes neuronales funcionan tan bien (similar a la ya existente para las SVM). Según Vapnik, padre de las SVM y firme defensor de su criatura, nadie en su sano juicio seguiría utilizando redes neuronales en el año 2005, todo el mundo usaría SVM... Los dos perdieron su apuesta y el único que cenó gratis fue Yann LeCun.

- *Datos*

Hoy en día se dispone de conjuntos de datos de un tamaño inimaginable para los que trabajaron con la generación anterior de redes neuronales, en los años 80 del siglo XX. Según Cisco Systems, el tráfico global en Internet era de apenas 100GB al día en 1992. En 2015, ese tráfico de datos se había multiplicado por diecisiete millones hasta los 20,235GB por segundo. Mientras que hace décadas la elaboración de conjuntos de datos con los que entrenar una red neuronal podía ser una ardua tarea, hoy en día resulta relativamente simple recopilar millones de imágenes, cientos de horas de vídeo o miles de horas de grabaciones de voz con los que entrenar redes neuronales artificiales. Los programadores, en vez de tener que diseñar algoritmos a medida, pueden recurrir a algoritmos de aprendizaje, genéricos, a los que les suministran terabytes de datos para que sea el ordenador el que se encargue de descubrir por sí mismo cómo reconocer objetos o palabras.

- *Potencia de cálculo*

Los coprocesadores matemáticos para realizar operaciones aritméticas existen, para ordenadores personales, desde el 8087 que complementaba al microprocesador 8086 en 1980. En 1996, Intel incluyó el juego de instrucciones MMX para sus procesadores Pentium. Este juego de instrucciones SIMD [*Single Instruction, Multiple Data*] permitía al procesador ejecutar la misma operación sobre varios datos en paralelo, algo muy útil para acelerar cálculos de tipo geométrico en dos y tres dimensiones. MMX sería posteriormente extendido por SSE [*Streaming SIMD Extensions*] para los procesadores Pentium III en 1999 y AVX [*Advanced Vector Extensions*] para los procesadores Sandy Bridge en 2011.

En paralelo al desarrollo de instrucciones SIMD para microprocesadores, surgió una industria dedicada a la comercialización de tarjetas gráficas. Especialmente orientadas hacia los “jugones” de videojuegos, incluyen circuitos optimizados para la creación de las imágenes y los efectos especiales típicos de la industria del entretenimiento digital. Aunque en los años 90 ya existían empresas establecidas como S3, ahora parte de HTC, o ATI [*Array Technology Inc.*], adquirida en 2006 por AMD, fue NVIDIA la que acuñó el término GPU [*Graphical Processing Unit*] en 1999 con su GeForce 256. Las GPU modernas tienen una arquitectura altamente paralela, similar a la de los supercomputadores vectoriales de antaño, como el Cray-1 de 1976. Un supercomputador Cray-1, que costaba 9 millones de dólares de entonces, era capaz de ejecutar 160 millones de operaciones en coma flotante por segundo (160 MFLOPS), dos por cada ciclo de su reloj a 80MHz. Una GPU actual, NVIDIA GTX 1080, de 2016, es capaz de ejecutar 8 billones de operaciones por segundo (8 TFLOPS), cincuenta mil veces más.

La extensión AVX-512 para 512 bits tiene prevista la inclusión de instrucciones VNNIW [*Vector Neural Network Instructions Word, variable precision*], instrucciones vectoriales específicamente diseñadas para Knights Mill, el código que usa Intel para sus futuros procesadores Xeon Phi especializados en *deep learning*.

Además, se puede adquirir por menos de mil dólares y consume más o menos lo que un pequeño electrodoméstico (180W).

La arquitectura paralela de una GPU la hace más eficiente que una CPU de propósito general para implementar algoritmos que procesen grandes bloques de datos en paralelo. La gente no tardó demasiado en darse cuenta de que la potencia de cálculo de una GPU se podía aprovechar más allá del mundo de los videojuegos, algo a lo que también contribuyó la estandarización, en 2007, del API utilizado para programar las GPU NVIDIA, denominado CUDA [*Compute Unified Device Architecture*]. Las redes neuronales están formadas por grandes cantidades de neuronas que operan en paralelo. Con su estructura regular y sus cálculos fácilmente paralelizables, han sido las grandes beneficiadas de la irrupción de GPU económicas dotadas con una potencia de cálculo reservada, hasta hace poco, a los grandes supercomputadores del Top 500. Lo que antes requería semanas en una CPU, ahora se puede hacer en horas con ayuda de una GPU. Y también se comercializan sistemas aún más potentes, que incluyen múltiples GPU en paralelo, tales como NVIDIA DGX-1 o Facebook Big Sur. Eso sí, se trata de sistemas que ya no están al alcance de todos los bolsillos.

■ *Algoritmos*

Como ya comentamos anteriormente, en 2006 se propusieron diversas técnicas que permitían lograr con éxito lo que hasta entonces se había considerado inviable en la práctica: entrenar redes neuronales con múltiples capas ocultas. Geoffrey Hinton, Yoshua Bengio, y Yann LeCun presentaron formas de afinar el entrenamiento de redes neuronales utilizando algoritmos *greedy* que iban ajustando los parámetros de una red ‘profunda’ capa por capa.

El uso de la expresión *deep learning* en redes neuronales artificiales tiene su origen en el año 2000,¹⁰⁴ aunque fue otro artículo de Geoffrey Hinton y Ruslan Salakhutdinov, publicado en 2006 por la revista Science,¹⁰⁵ el que sirvió para rebautizar las redes neuronales artificiales y despertar de nuevo el interés en ellas, más allá de los investigadores irreductibles que habían seguido trabajando en ellas cuando casi nadie les prestaba atención. Desde entonces, se hace referencia a las redes neuronales artificiales con un apelativo mucho más comercial: *deep learning*.

Limitaciones del deep learning

Las limitaciones que impidieron que las técnicas de *deep learning* se desarrollaran antes eran más de tipo práctico que teórico. Ni los ordenadores eran lo suficientemente potentes, ni los conjuntos de datos lo suficientemente grandes. Una vez solventadas esas dificultades prácticas,

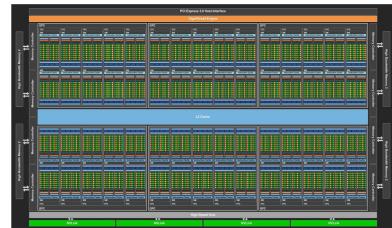


Figura 25: Arquitectura paralela de una GPU NVIDIA Pascal, compuesta por 6 clusters GPC [Graphics Processing Cluster] y hasta 60 multiprocesadores SM [Streaming Multiprocessors], cada uno con 64 núcleos CUDA. El chip dispone de 8 controladores de memoria de 512 bits (4096 bits en total) y 15,300 millones de transistores FinFET de 16 nm en 610 mm². Fuente: NVIDIA.

<https://www.top500.org/>

¹⁰⁴ Igor N. Aizenberg, Naum N. Aizenberg, y Joos P. Vandewalle. *Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications*. Kluwer Academic Publishers, 2000. ISBN 0792378245

¹⁰⁵ Geoffrey E. Hinton y Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006. DOI: 10.1126/science.1127647

Ruslan ‘Russ’ Salakhutdinov es otro de los doctorandos de Hinton, profesor en la Universidad de Carnegie-Mellon, Pittsburgh, Pennsylvania, y director de investigación en I.A. de Apple.

los fundamentos formales que ya se conocían y empleaban en entornos limitados desde los años 80 permitieron el despegue del *deep learning*.

No obstante, eso no quiere decir que el uso de redes neuronales artificiales esté por completo exento de problemas. Sus dos aspectos más problemáticos son los derivados del sobreaprendizaje y su carácter de cajas negras. Para resolver el primero, que es un problema común con otros muchos modelos de aprendizaje automático, se han propuesto multitud de técnicas que, más o menos, nos permiten soslayarlo. En cuanto al segundo, es algo sobre lo que todavía queda mucho por hacer y que puede tener implicaciones con respecto a la seguridad de los sistemas que emplean redes neuronales internamente.

Problemas derivados del sobreaprendizaje

El sobreaprendizaje se produce siempre que un modelo se ajusta tan bien a su conjunto de entrenamiento que deja de generalizar correctamente cuando lo utilizamos sobre un conjunto de prueba diferente al conjunto de datos de entrenamiento que utilizamos para construirlo. Es un problema habitual en muchas técnicas de aprendizaje automático.

Cuando utilizamos redes neuronales, las redes neuronales incluyen multitud de parámetros ajustables: los pesos que modelan las conexiones entre neuronas. Ese número elevado de parámetros las hace propensas a sufrir problemas de sobreaprendizaje [*overfitting*]. De hecho, hay quien piensa que es su principal inconveniente.

El conjunto de entrenamiento contiene las regularidades que nos permiten construir un modelo de aprendizaje automático. Sin embargo, también incluye dos tipos de ruido: errores en los datos y errores de muestreo. Los errores en los datos son inevitables y provienen del proceso de adquisición de datos, especialmente si éste incluye algún tipo de procesamiento manual. Las etiquetas asociadas a los casos de entrenamiento pueden ser erróneas, por ejemplo, en un porcentaje nada desdeñable de éstos. Por otro lado, el conjunto de datos, por grande que sea, no deja de ser una muestra de una población mayor. En él existirán regularidades accidentales que no se deben al problema en sí que deseamos modelar, sino a los casos particulares que acaban formando parte de nuestro conjunto de datos. Cuando ajustamos un modelo a esos datos, nunca podemos saber si estamos aprovechando las regularidades “reales” de los datos o si nuestro modelo está identificando las regularidades accidentales debidas al error de muestreo. Cualquier modelo mezclará ambas. Si, como sucede con las redes neuronales, nuestro modelo es extremadamente flexible, entonces será capaz de modelar todos los matices del conjunto de datos de entrenamiento, tanto esenciales como accidentales. Cuando el modelo ajusta los rasgos accidentales, pierde capacidad de generalización y, en la práctica, esto puede resultar desastroso.

En términos generales, un modelo de aprendizaje con muchos parámetros, como es el caso de las redes neuronales, será capaz de ajustarse mejor a los datos que le proporcionemos, aunque no resulte demasiado económico. En términos de la descomposición del error en sesgo y varianza, diríamos que el modelo exhibe una varianza elevada (si tuviese muy pocos parámetros hablaríamos de sesgo). No debería sorprendernos, por tanto, que un modelo con muchos parámetros se ajuste demasiado bien a los datos de entrenamiento, hasta el punto de sobreaprender. ¿Qué estrategias generales podemos utilizar para evitar ese sobreaprendizaje? La primera opción sería conseguir más datos, la estrategia recomendada si disponemos de capacidad de cálculo para entrenar la red usando más datos. Una segunda opción sería ajustar la capacidad del modelo, intentando que ésta sea suficiente para ajustar las regularidades reales pero no las espurias (asumiendo, sin justificación alguna, que éstas serán más débiles que las auténticas). Una tercera opción es combinar múltiples modelos, ya sean modelos de diferentes características o modelos del mismo tipo entrenados con diferentes muestras del conjunto de entrenamiento (p.ej. *bagging*).

Afortunadamente, además de las estrategias generales que podemos utilizar para cualquier modelo de aprendizaje automático, se han propuesto multitud de técnicas específicas que se pueden emplear para prevenir y atenuar los efectos del sobreaprendizaje en redes neuronales artificiales. Éstas incluyen sencillas heurísticas, como el decaimiento de pesos [*weight decay*], que penaliza la presencia de pesos elevados en la red, o el uso de pesos compartidos [*weight sharing*], que reduce el número de parámetros de una red neuronal, con lo que se reduce su capacidad y, por tanto, la posibilidad de sufrir sobreaprendizaje. Otras técnicas de prevención del sobreaprendizaje se incorporan en el propio algoritmo de entrenamiento de la red neuronal, como la parada temprana [*early stopping*], que detiene el proceso iterativo de aprendizaje en el momento en el que se detecta un aumento de la tasa de error en un conjunto de validación independiente del conjunto de entrenamiento. Otra posibilidad más consiste en la combinación de múltiples modelos, ya sea de forma explícita mediante la creación de ensambles (v.g. *model averaging*) o de forma implícita mediante la incorporación de técnicas que son, desde un punto de vista formal, equivalentes al uso de múltiples modelos (v.g. *dropout*). Hay incluso quien aboga por un proceso de pre-entrenamiento previo, realizado de forma no supervisada, para pre-ajustar los parámetros de la red antes de someterla a un entrenamiento de tipo supervisado, lo que puede servir para prevenir el sobreaprendizaje a la vez que resulta útil en situaciones en los que disponemos de muchos más datos no etiquetados (para el pre-entrenamiento no supervisado) que etiquetados (para el entrenamiento supervisado).

Como vemos, existe una amplia gama de técnicas y herramientas a

nuestra disposición para prevenir el sobreaprendizaje al que son propensas las redes neuronales. Dada esa propensión, la estrategia habitual en el entrenamiento de redes neuronales artificiales es permitir que la red sobreaprenda y luego regularizar. En Estadística, se denomina regularización a cualquier proceso mediante el cual se incorpora información adicional para prevenir el sobreaprendizaje, ya sea un término adicional en la función de coste que se pretenda minimizar (como en *weight decay*), un mecanismo de control durante el proceso de entrenamiento (*early stopping*) o la anulación selectiva de partes de una red neuronal para prevenir su coadaptación (*dropout*).

Para algunos, como Geoffrey Hinton, el sobreaprendizaje no es más que una ilusión de los frecuentistas. Un frecuentista diría que, si no dispone de demasiados datos, debería utilizar un modelo más simple, porque uno más complejo sobreaprenderá. Asumimos que, al aprender, el modelo elegirá el mejor conjunto de parámetros posible, aquél que maximice la probabilidad de los datos dados los parámetros del modelo $p(\text{datos}|\text{parámetros})$. Es lo que hace un estimador de máxima verosimilitud [*maximum likelihood*]. Cuando disponemos de pocos datos, muchas combinaciones diferentes de parámetros nos darán una probabilidad similar, de ahí la posibilidad de sobreaprendizaje, al no ser capaz de distinguir unas de otras. Si dispusiésemos de una estimación más precisa de la distribución $p(\text{datos}|\text{parámetros})$, el sobreaprendizaje desaparecería. Un bayesiano diría que no existe razón alguna por la que la cantidad de datos disponible deba influir en nuestras creencias a priori sobre la complejidad del modelo. Como tampoco debería influir la presencia de ruido en los datos. Si partimos de una creencia adecuada a priori, dada por la verosimilitud $p(\text{datos}|\text{parámetros})$, deberíamos obtener predicciones razonables si ajustamos la distribución a posteriori, $p(\text{parámetros}|\text{datos})$, algo que se consigue aplicando el teorema de Bayes.

En definitiva, según Hinton, si su red neuronal no está sobreaprendiendo, el problema es suyo y debería estar utilizando una red más grande, con mayor capacidad: una red que sea capaz de sobreaprender dados los datos de entrenamiento. En *deep learning*, si la red no sobreaprende, no lo está haciendo bien.

Las redes neuronales como cajas negras

En determinados dominios de aplicación, un problema más acuciante de las redes neuronales artificiales es nuestra incapacidad para determinar cómo llegan las redes neuronales a una conclusión. En una red neuronal, podemos observar la entrada de la red y ver cuál es su salida, pero su funcionamiento interno es algo que no se puede describir de forma simbólica. Puede que apreciemos indicios que nos permitan intuir qué variables influyen más en una decisión determinada, pero nunca de una

A la hora de interpretar probabilidades, los estadísticos bayesianos lo hacen como el resto de los mortales: como una indicación de la plausibilidad de un evento. Los frecuentistas, en cambio, las interpretan como las frecuencias de un evento a largo plazo. Si no sabemos dónde hemos dejado un teléfono inalámbrico, el localizador de la base del teléfono hará que éste comience a sonar. Un frecuentista escucha el sonido y, con un mapa mental de su casa, identifica la zona donde puede estar el teléfono. Un bayesiano, aparte del mapa mental de su casa, también tiene en cuenta los sitios donde habitualmente deja olvidado el teléfono. Para determinar dónde buscar, el bayesiano combina la evidencia del sonido con su conocimiento a priori de sitios donde dejó el teléfono en el pasado. El frecuentista dirá que el conocimiento a priori ya estaba incorporado en su modelo. La distinción es más filosófica que práctica.

forma demasiado precisa. Para nosotros, una red neuronal es, en gran medida, una caja negra.

En situaciones en la que no sólo sea importante la validez del modelo, sino su credibilidad, puede que el carácter opaco de una red neuronal juegue en nuestra contra. La validez de un modelo la podemos evaluar de múltiples formas, p.ej. con una validación cruzada. Las pruebas que realizemos nos pueden indicar que, en efecto, una red neuronal ofrece mejores resultados que otras técnicas de aprendizaje automático. Sin embargo, su credibilidad es otra historia. Al tratarse de una faceta subjetiva, vinculada a la percepción que otros pueden tener de su fiabilidad, es algo que no podemos avalar con argumentos meramente cuantitativos. Por desgracia, dado que no podemos explicar su funcionamiento en detalle, tampoco podemos recurrir a los argumentos que utilizaríamos para defender un modelo simbólico, en el que su razonamiento es fácilmente interpretable. Esto puede hacer que muchos posibles usuarios se muestren reacios a adoptar el uso de redes neuronales artificiales.

Algunas de las técnicas de aprendizaje automático más versátiles funcionan de forma algo misteriosa. Las decisiones automatizadas con ayuda de redes neuronales o máquinas de vectores de soporte son, en gran parte, completamente inescrutables. Aunque funcionen realmente bien, si son incapaces de ofrecer una explicación, puede que no nos dejen usarlas. Tal vez porque alguien se sienta inseguro al no poder justificar una decisión en caso de que dicha decisión se demuestre errónea en el futuro. Tal vez porque políticos miopes, ignorantes del tema, decidan dejar este tipo de herramientas en un limbo legal al ser incapaces de entender su funcionamiento (o, más bien, no intentar comprenderlo). El desarrollo futuro de estas técnicas de aprendizaje puede resultar más frágil de lo que parece.

En entornos altamente regulados, como las finanzas, la ley puede obligar a una compañía a que explique las razones tras una decisión tomada con ayuda de un modelo automatizado. ¿Por qué se le deniega la concesión de un crédito a un posible cliente? ¿Por qué se le incrementa el importe de la póliza de un seguro médico? Si esas obligaciones se generalizan, podríamos presenciar pleitos ocasionados por los anuncios que se nos muestran al navegar por Internet o las recomendaciones personalizadas que recibimos. ¿Por qué se le ofrecen productos para bebés a una adolescente? ¿Por qué nos recomiendan un tipo de música y no otro? ¿Existe algún sesgo de tipo ideológico, sexista o racista que esté influyendo en la información que recibimos? Cuando los reguladores extremen su celo en la “responsabilidad algorítmica”, muchas de las técnicas de aprendizaje automático actuales no se podrán seguir utilizando tal cual. Probablemente, surjan nuevas técnicas que lo que hagan sea, una vez tomada una decisión, inventar una explicación plausible que sirva para cubrir el expediente legal. Al fin y al cabo, encontrar una

Recordemos que la Unión Europea pretende implantar, en 2018, una regulación general de protección de datos según la cual los usuarios tendrían el derecho a recibir una explicación de las decisiones tomadas de forma algorítmica. Algo a lo que los burócratas de Bruselas denominan “responsabilidad algorítmica”. Una red neuronal o, ya puestos, una SVM o casi cualquier sistema de recomendación actual, seguramente sería incapaz de ofrecer una explicación al gusto de la burocracia vigente. Al menos, directamente...

Esto es algo que ya sucedió en Estados Unidos hace un tiempo. Target, una cadena de grandes almacenes, comenzó a utilizar herramientas de minería de datos para personalizar las ofertas que diferentes clientes recibían en función de su comportamiento. Cuando envió cupones de productos para bebés a una adolescente de Minneapolis, un airado padre se quejó de que pretendiesen que su hija se quedase embarazada. Unos días después, llamó por teléfono para pedir disculpas. El modelo predictivo utilizado por Target descubrió el embarazo antes que el padre.

La Ley de Hiram: “Si se consultan suficientes expertos, se puede confirmar cualquier opinión”.

justificación a una decisión ya tomada es un problema mucho más sencillo que descubrir un modelo que tome la decisión adecuada en primer lugar. Existen infinitas explicaciones, más o menos plausibles, que podrían utilizarse para justificar una decisión algorítmica. Y no tendríamos que llegar al extremo del mal estudiante que hace responsable a su perro de haberse comido el papel que contenía sus deberes.

En la práctica, ¿cómo podemos conseguir que un sistema basado en redes neuronales proporcione una justificación a las salidas que genera? Existen diferentes estrategias que podrían funcionar. Podríamos buscar algunos casos del conjunto de entrenamiento que sirviesen de ejemplo, a modo de breve explicación basada en casos particulares. En un texto, podríamos resaltar algunas palabras clave o fragmentos completos del texto. Sobre una imagen, podríamos enmarcar las zonas más significativas de la imagen, de forma que podamos localizar los factores clave de una decisión. El inconveniente de todas esas alternativas, obviamente, es que la explicación no dejará de ser una simplificación de la realidad que se oculta tras la decisión real de la red neuronal.

El problema no es exclusivo de las técnicas de aprendizaje automático. Nosotros, humanos, tampoco somos siempre capaces de explicar cómo tomamos decisiones. A menudo, evaluamos una situación de forma intuitiva.¹⁰⁶ Incluso cuando alguien nos da una justificación medianamente razonable, su exposición de argumentos será, más que probablemente, una explicación incompleta. Puede que se deba a que nuestra propia inteligencia no sea por completo racional. Parte de ella es meramente instintiva, subconsciente y tan inescrutable como las redes neuronales artificiales.

Hasta ahora, nunca habíamos construido máquinas que funcionasen de forma que sus creadores no pudiesen comprender. Ruslan Salakhutdinov, director de investigación en Inteligencia Artificial en Apple y profesor de Carnegie Mellon University, considera que la inclusión de inteligencia social en las técnicas de aprendizaje automático será esencial para el desarrollo de nuevas formas de interacción entre los seres humanos y las máquinas. Éstas tendrán que saber cómo inspirar confianza, de forma similar a como las convenciones sociales nos permiten saber qué esperar de otros. Nosotros tendremos que ser cautos, igual que lo somos con desconocidos. Si una máquina no es mejor que nosotros al explicar qué está haciendo, Salakhutdinov recomienda no confiar en ella.

Una explicación real del porqué de una decisión algorítmica puede ser del todo imposible en sistemas que utilicen *deep learning*, ya sea para clasificar imágenes, recomendar canciones, mostrar anuncios o tareas más serias. Las redes neuronales ajustan sus parámetros para completar con éxito una tarea, pero lo hacen de una forma que nosotros no podemos comprender desde el punto de vista simbólico. Podemos entender los algoritmos con los que se realiza ese ajuste durante el entrenamiento de

¹⁰⁶ Daniel Kahneman. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011. ISBN 0374275637

El filósofo Daniel Dennett sugiere que la creación de sistemas que sean capaces de realizar tareas que sus creadores no saben cómo hacer forma parte consustancial de la evolución de la inteligencia en sí misma. Éste sería el punto crítico en el que, según algunos, nos encontramos actualmente.

la red, pero generalmente seremos incapaces de determinar las causas detalladas de una decisión particular cuando la red ya está entrenada. En aplicaciones militares de defensa o de seguridad, este hecho puede suponer un hándicap, especialmente a posteriori si se producen accidentes y se busca al responsable (o, tal vez, a un cabeza de turco). ¿Por qué se pilota un vehículo autónomo de esa forma y no de otra? ¿Qué criterios se utilizan para identificar posibles objetivos? ¿Por qué se señala a alguien particular como sospechoso de terrorismo si, en principio, parece llevar una vida completamente normal? ¿Es un caso flagrante de discriminación por edad, sexo, raza o religión?

Dado que los sistemas de *deep learning* no toman decisiones de la misma forma que nosotros, esto ocasiona la aparición de problemas de seguridad en todos aquellos ámbitos donde se recurre a ellos:

- Dada la importancia que tienen los sistemas de recomendación en muchos aspectos de nuestra vida *online*, esos sistemas han sido objeto de numerosos ataques que pretenden manipular las recomendaciones que ofrecen. En ocasiones, para promover un artículo determinado, haciendo que se recomiende más a menudo. En otras ocasiones, para tumbar su reputación [*nuke*] de forma que nunca se recomienda. Denominados ataques por complicidad [*shilling attacks*], los incentivos económicos para los atacantes convierten el diseño de sistemas de recomendación en una guerra de desgaste sin fin (como la de los fabricantes de antivirus con los desarrolladores de nuevos tipos de *malware*).

Estamos en una situación en la que los sistemas automáticos basados en redes neuronales nos pueden ayudar en la construcción de mejores sistemas de recomendación, a la vez que dificultan la interpretación de los resultados que proporcionan e interfieren en nuestra capacidad de detectar manipulaciones.

- Las redes neuronales artificiales han demostrado ser especialmente útiles en el reconocimiento de patrones complejos, como puede ser la identificación de objetos en imágenes. Se ha demostrado que se puede engañar a una red neuronal para que crea percibir cosas que no están realmente en la imagen. Es más, se ha conseguido la creación de imágenes que, siendo indistinguibles de las originales para nosotros, confunden por completo a una red neuronal.

Si dotamos a un vehículo autónomo de una cámara con la que interpretar señales de tráfico y semáforos, pensemos en las vulnerabilidades de seguridad de un sistema así cuando la red neuronal es la encargada de determinar el significado de una señal de tráfico y el coche autónomo actúa de acuerdo a la predicción realizada por la red neuronal.¹⁰⁷

Y no hemos siquiera mencionado los problemas de seguridad que se



Figura 26: Una señal de tráfico en un cruce sin visibilidad. Nosotros la vemos como una señal de stop pero... ¿la verá igual una red neuronal? ¿Puede que alguien la haya manipulado para que una red neuronal la interprete incorrectamente sin que nosotros nos demos cuenta? Se trata de un problema para el que no existe solución.

¹⁰⁷ Patrick McDaniel, Nicolas Papernot, y Z. Berkay Celik. Machine learning in adversarial settings. *IEEE Security & Privacy*, 14(3):68–72, 2016. ISSN 1540-7993. DOI: 10.1109/MSP.2016.51

derivan de que un sistema autónomo no sea 100 % fiable. No sólo es difícil predecir bajo qué circunstancias un sistema autónomo cometerá errores. Sabemos que, inevitablemente, lo hará. Sin embargo, desconocemos cuándo. Si circulamos en un vehículo semi-autónomo, la presencia de falsas alarmas puede hacernos bajar la guardia tanto como su ausencia por completo, que podría llegar a crear una falsa sensación de seguridad en nosotros. Si un sistema con una autonomía limitada funciona perfectamente durante el 99.99 % del tiempo, tendríamos problemas en situaciones extraordinarias que requiriesen la intervención humana. El conductor podría haberse distraído por completo, confiado en la fiabilidad del sistema, y no reaccionar a tiempo para prevenir un accidente inminente.

Inspiración biológica

Pese a sus inevitables limitaciones, comunes por otro lado con la gran mayoría de técnicas de aprendizaje automático, los algoritmos de *deep learning* son de los más robustos. Se adaptan con facilidad a distintos tipos de datos, se entrenan con algoritmos relativamente sencillos y escalan bien a grandes conjuntos de datos. Son, a día de hoy, una de las herramientas más versátiles de las que disponen los científicos de datos a la hora de detectar patrones en conjuntos de datos. Las redes neuronales son capaces de identificar y extraer las características que son realmente relevantes y deben utilizarse para resolver un problema, algo que escapa de las posibilidades de otras técnicas de aprendizaje automático.

Para los conexionistas, los modelos de *deep learning* son la respuesta a todos los problemas de la I.A.. Para los demás, son una potente herramienta, excelente a la hora de reconocer patrones. Son, incluso mejores que nosotros, los humanos, en reconocimiento de voz o clasificación de imágenes. Sin embargo, están algo limitados a la hora de razonar, pese a que se han comenzado a realizar propuestas que extienden las redes neuronales artificiales en ese sentido. Las máquinas de Turing neuronales [NTM: *Neural Turing Machines*] intentan combinar las capacidades de reconocimiento de patrones de una red neuronal con las capacidades algorítmicas de un ordenador programable.¹⁰⁸ Las computadoras neuronales diferenciales [DNC: *Differentiable Neural Computers*] añaden a las NTM mecanismos de atención que controlan qué parte de la memoria se activa en cada momento.¹⁰⁹

Pero lo que hace realmente interesantes a las redes neuronales, tanto a los especialistas en I.A. como a los profanos en el tema, es su inspiración biológica. No es un rasgo específico de las redes neuronales artificiales, ya que muchas de las técnicas de *soft computing* o Inteligencia Computacional están inspiradas en la naturaleza. La naturaleza siempre ha servido como fuente de inspiración para ingenieros y científicos, por lo que no debería

¹⁰⁸ Alex Graves, Greg Wayne, y Ivo Danihelka. Neural Turing Machines. *arXiv e-prints*, arXiv:1410.5401, 2014. URL <http://arxiv.org/abs/1410.5401>

¹⁰⁹ Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, y Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, October 2016. ISSN 0028-0836. DOI: 10.1038/nature20101

extrañarnos que muchas de las técnicas de aprendizaje automático usadas en I.A. estén basadas en cómo aprenden los seres vivos, con alguna que otra licencia por parte del diseñador de algoritmos de aprendizaje.

Se pueden aducir varias razones de peso para estudiar el aprendizaje en los seres vivos. En primer lugar, como método de comprensión del proceso de aprendizaje, desde el punto de vista de la Psicología. En segundo lugar, aunque no por ello sea menos interesante, para conseguir sistemas que aprendan de forma automática, desde una perspectiva más propia de la Inteligencia Artificial. Para un animal, el aprendizaje a partir de la experiencia es muy importante, ya que le permite no volver a cometer los mismos errores una y otra vez. No en vano, la capacidad de adaptarse a nuevas situaciones y resolver problemas se considera una característica clave de la inteligencia de los seres vivos.

Si nos centramos en el estudio de las redes neuronales, algunos científicos construyen modelos para intentar comprender el funcionamiento del cerebro (los científicos construyen para estudiar). Ése es el foco de actividad de la denominada neurociencia computacional. Otros científicos e ingenieros se inspiran en el cerebro para intentar diseñar mecanismos que nos permitan avanzar en la construcción de sistemas inteligentes (los ingenieros estudian para construir). Aquí se enmarcaría el trabajo de los especialistas en *deep learning*, que suelen interpretar libremente los descubrimientos neurocientíficos y elaboran modelos que, aunque no resulten biológicamente plausibles, ayudan a resolver problemas prácticos concretos. Al fin y al cabo, para que un avión vuela, no tiene por qué mover las alas como un pájaro.

Aquí nos volvemos a encontrar con una de las dicotomías que dividen a la comunidad de especialistas en Inteligencia Artificial. Por un lado, simbólicos y bayesianos opinan que las máquinas deberían razonar de acuerdo a reglas explícitas, lógicas o probabilísticas, de forma que su comportamiento sea completamente transparente para cualquiera que le dedique el tiempo suficiente a examinar el código que dirige su funcionamiento. El resultado sería siempre una I.A. basada en modelos interpretables cuyo funcionamiento se puede explicar.

Por otro lado, evolutivos y conexionistas consideran que la inteligencia puede emerger si las máquinas aprenden observando y experimentando, tal como sucede con los seres vivos en la naturaleza. Inspirados en la biología, los modelos evolutivos y conexionistas pretenden que sea el ordenador el que se programe a sí mismo. En vez de que el programador especifique el algoritmo que resuelve un problema, el ordenador es capaz de generar su propio ‘algoritmo’ a partir de datos de ejemplo y cierto grado de supervisión mediante el que se le indica cuál es el resultado deseado. Las técnicas bioinspiradas le dan la vuelta a la programación de máquinas inteligentes. En vez de programar el algoritmo que les permita razonar adecuadamente, se programa un algoritmo de aprendizaje que

les permita a las máquinas descubrir por sí mismas cómo resolver un problema.

Simbólicos y bayesianos no creen en la emulación de la naturaleza para obtener sistemas inteligentes, a diferencia de evolutivos y conexionistas. Tanto los simbólicos como los bayesianos pretenden descubrir los principios que subyacen al aprendizaje y, por tanto, a la inteligencia (humana incluida). Es el típico enfrentamiento entre teorías descriptivas y teorías prescriptivas o normativas que se da en otras ramas del conocimiento, desde la filosofía a la lingüística. La diferencia entre el 'así son las cosas y así se las hemos contado' de los descriptivos y el 'así debería ser' de los normativos. No sería justo desdeñar los esfuerzos de simbólicos y bayesianos. Descubrir cómo se debería aprender puede ayudarnos a comprender cómo lo hacemos, ya que ambas cosas están presumiblemente relacionadas. A fin de cuentas, los comportamientos esenciales para nuestra supervivencia han evolucionado durante millones de años y su descripción puede que no difiera demasiado de la prescripción normativa.

Dentro de las técnicas bioinspiradas, conexionistas y evolutivos difieren a la hora de abordar el problema. En el fondo, responden de forma diferente a la siguiente pregunta: la inteligencia, ¿nace o se hace?

Ante la siguiente pregunta: ¿Qué mecanismo es mejor para construir sistemas inteligentes?

- Un conexionista dirá que el mejor mecanismo conocido de resolución de problemas es el cerebro humano, que inventó "la rueda, Nueva York, las guerras..."¹¹⁰ y todo lo demás.
- Un evolutivo responderá que el mejor mecanismo es la evolución, que creó el cerebro humano.

Ambos se inspiran en la naturaleza, si bien los evolutivos pretenden descubrir la estructura adecuada para un problema dado mientras que los conexionistas suelen centrarse en ajustar los parámetros de una estructura predefinida para resolver un problema de forma óptima. Igual que nosotros somos el resultado de nuestra naturaleza y de nuestra experiencia, nacemos y nos hacemos, puede que la clave de la consecución de sistemas inteligentes sea una combinación adecuada de evolución y entrenamiento.

El éxito actual de las redes neuronales utilizadas en *deep learning* se debe fundamentalmente a tres características bioinspiradas: su plasticidad, que les permite adaptarse y aprender; su organización jerárquica, que les faculta para resolver problemas complejos por composición; y, por último, su forma de modelar la percepción, que les proporciona mecanismos con los que resolver problemas de reconocimiento de patrones en señales de voz (reconocimiento de voz) y en imágenes visuales (identificación de objetos en visión artificial). Analicémoslas con algo más de detalle...

¹¹⁰ Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Pan Books, 1979.
ISBN 0330258648

Plasticidad

Las técnicas de entrenamiento utilizadas en *deep learning* ajustan los parámetros de modelos de redes neuronales en los que cada neurona no es más que una función matemática que combina una serie de entradas para generar una salida. La función se suele mantener fija y la red aprende mediante el ajuste de los pesos asociados a las entradas. Aunque el funcionamiento de una neurona es sencillo, es la combinación de muchas neuronas interconectadas lo que hace inescrutable el comportamiento de una red neuronal en su conjunto.

El ajuste de los pesos de una red neuronal artificial se inspira en la forma en que aprenden las redes neuronales biológicas. Una red neuronal no nace preprogramada con todo lo que un organismo necesita para sobrevivir, sino que es capaz de aprender incorporando información a las sinapsis que conectan unas neuronas con otras. En 1949, Donald Hebb, un psicólogo canadiense que entonces trabajaba en la Universidad McGill de Montreal, propuso una teoría sencilla de ese aprendizaje: “Cuando el axón de una célula A está lo suficientemente cerca de una célula B y repetida o persistentemente colabora en su activación, se produce algún proceso de crecimiento o cambio metabólico sobre una o ambas células mediante el cual se incrementa la eficiencia de A como una de las células que activa B”.¹¹¹ Hebb hipotetizó entonces que, durante el aprendizaje, la superficie de la unión sináptica aumentaba. Teorías más recientes atribuyen el aumento de la efectividad de una sinapsis a una mayor liberación de neurotransmisores en la neurona presináptica.

La plasticidad de una red neuronal biológica no se limita a cambios en las conexiones sinápticas con las que se esculpe su circuitería. La actividad neuronal generada durante la experiencia vital de un animal se traduce también en la reserva de más espacio en el córtex cerebral para procesar la información relevante. Es decir, la actividad neuronal influye en la conectividad neuronal y también en la cantidad de neuronas dedicadas a tareas específicas. Durante la maduración de un individuo, las regiones cerebrales más activas se expanden relativamente más que las menos activas. Por ejemplo, los neurocientíficos han descubierto que los taxistas de Londres acaban teniendo una mayor densidad de neuronas en el hipocampo, un área del cerebro asociada a la navegación espacial. Sin embargo, seamos diestros o zurdos, estudios que han intentado medir la extensión de la zona del cerebro humano dedicada a las manos no encontraron diferencias significativas entre el espacio cortical dedicado a la mano derecha y a la mano izquierda.

Las experiencias iniciales de un individuo también pueden determinar la forma en la que se cablea su córtex cerebral. Hasta el punto de que experiencias posteriores no pueden revertir ese cableado inicial. Por ejemplo, la experiencia visual determina cómo se conectan las distintas

¹¹¹ Donald O. Hebb. *The Organization of Behavior*. John Wiley, New York, 1949

partes del córtex visual en el cerebro. Cuando se mantiene tapado uno de los ojos de un gato durante sus primeras semanas de vida, su córtex visual se adapta al ojo por el que sí percibe estímulos visuales y, aunque destapemos su ojo, el gato queda tuerto de por vida. No sólo eso. Si, nada más nacer, se cruzan los nervios ópticos y auditivos de un animal, de forma que el nervio óptico conecte con la zona auditiva del córtex y el nervio auditivo conecte con la zona que habitualmente recibe señales visuales en el córtex, el animal es capaz de ver y oír: su córtex ‘visual’ aprenderá a interpretar señales auditivas y su córtex ‘auditivo’ descifrará las señales visuales recibidas a través del nervio óptico.

También se ha observado que la plasticidad cortical disminuye cuando un individuo madura. La disminución gradual de la plasticidad cortical que padecemos, tanto los humanos como otros animales, proporciona una base neurobiológica a la observación de que aprendemos idiomas, música y convenciones sociales mucho mejor cuando somos niños que cuando somos adultos. El mismo fenómeno explica que un perro viejo sea incapaz de aprender nuevos trucos (con acento sureño: “*an ol’ dog learns no new tricks!*”).

La base biológica del aprendizaje, pues, parece estar en la creación y el fortalecimiento de conexiones entre neuronas. Se pueden crear nuevas conexiones entre neuronas ya conectadas o entre neuronas que, a priori, no estaban conectadas, incluso en individuos adultos (plasticidad estructural). Se puede fortalecer una conexión ya existente entre dos neuronas que se activan conjuntamente, mediante un proceso bioquímico conocido como potenciación a largo plazo (plasticidad sináptica). La forma técnica de hacer referencia al aprendizaje hebbiano. Y todo esto bajo las restricciones de consumo energético de las neuronas biológicas, las cuales tienen a mantener su actividad media en rangos que sean energéticamente asumibles para el organismo, para lo que modifican su excitabilidad (plasticidad intrínseca). Esto es así tanto en el cerebro humano como en el cerebro de otros animales. Nuestro cerebro es inusualmente grande, pero funciona y está construido de forma similar al de otros animales.

Lo que parece claro es que no puede existir aprendizaje si no hay memoria. En Psicología se utilizan modelos de memoria que distinguen una memoria a corto plazo [*STM, short-term memory*] y una memoria a largo plazo [*LTM, long-term memory*]. Es el modelo de los sistemas expertos, populares en los años 80: una memoria de trabajo donde se manipulan hechos (*STM*) y una base de conocimiento que contiene reglas (*LTM*). En el caso del cerebro humano, la memoria a corto plazo tiene una capacidad limitada: el famoso número mágico 7 ± 2 del psicólogo cognitivo George Miller.¹¹² Además, su persistencia es muy limitada, de 15 a 30 segundos si no se repite verbalmente su contenido (codificación acústica, incluso para la información visual). El almacenamiento de hechos en memoria a corto plazo es frágil y se puede perder fácilmente, por algún

¹¹² George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, March 1956. DOI: 10.1037/h0043158

tipo de distracción o por el simple paso del tiempo.

El aprendizaje real involucra la creación de memorias a largo plazo, para lo que el cerebro necesita una señal de aprendizaje. La forma más eficiente de señal de aprendizaje para cualquier sistema, artificial o biológico, es la proporcionada por el modelo de diferencias temporales usado en aprendizaje por refuerzo [*TD-learning*]. Este modelo compara la recompensa esperada con la recompensa observada. La diferencia se utiliza para actualizar la predicción. Cuando la recompensa observada es mayor que la esperada se refuerza el comportamiento que da lugar a esa recompensa. Cuando es menor, se desincentiva. Cuantitativamente, el error en la predicción de la recompensa se evalúa como una señal normalizada de contraste: la magnitud del error en la estimación de la recompensa dividido por la desviación estándar de su distribución de probabilidad.

Cuando un animal percibe que una señal anticipa una recompensa, su activación neuronal declina ante la recompensa en sí y aumenta para la señal premonitoria. Es el típico aprendizaje por asociación del condicionamiento clásico conocido por los famosos experimentos de los perros de Pavlov. El animal aprende a anticipar las consecuencias de algo para, de esa forma, estar prevenido ante lo que pueda pasar y mejorar su capacidad predictiva, algo que evolutivamente le beneficia en su lucha por la supervivencia.

¿Cómo se transmiten esas señales de aprendizaje en el cerebro?¹¹³ Los axones de las neuronas transmiten la señal eléctrica en el cerebro. Un solo axón puede entregar la señal a muchas zonas diferentes del cerebro, el córtex prefrontal, el hipocampo, el cuerpo estriado ventral y dorsal, la amígdala o el hipotálamo. Para ello, una neurona puede tener un árbol dendrítico con hasta 10^6 terminales con vesículas llenas de moléculas neurotransmisoras como la dopamina. En el cerebro humano, las señales de aprendizaje han de llegar a un gran volumen de tejido neuronal, para lo que se estima que cada hemisferio cerebral contiene del orden de 200,000 neuronas especializadas en transmitir dichas señales de aprendizaje. Esas señales eléctricas se distribuyen hasta llegar a sólo unos micrómetros de distancia de las sinapsis que han de aprender, donde se liberan de forma precisa los neurotransmisores químicos en el espacio extracelular. El transmisor se difunde unas micras más hasta alcanzar varias sinapsis cercanas. Esa difusión a pequeña escala permite una temporización lo suficientemente precisa: la difusión en el espacio extracelular de 3 a 10 micras se realiza en decenas de milisegundos. No importa que la difusión del transmisor, desde el punto de vista espacial, alcance varias sinapsis circundantes, ya que sólo las sinapsis ya activadas responderán a la señal de aprendizaje: la señal de aprendizaje se difunde a todas las sinapsis, pero sólo se fortalecerán las que han sido activadas de forma repetida. Tal como predijo Donald Hebb mucho antes de disponer de la tecnología

¹¹³ Peter Sterling y Simon Laughlin. *Principles of Neural Design*. MIT Press, 2015. ISBN 0262028700

necesaria para poder analizar el fenómeno a nivel molecular.

Dado que la señal de aprendizaje ha de resultar breve para que el aprendizaje sea efectivo, la sensación de bienestar a la que normalmente se asocia es, por necesidad, episódica. Esa sensación es lo que Sigmund Freud llamaría felicidad. Ese bienestar es agradable cuando el entorno genera pequeñas señales de error en la predicción de recompensas con cierta frecuencia. Si el entorno se hace demasiado previsible, las posibles fuentes de esas señales de error desaparecen. Nos aburrimos y los intervalos entre períodos de satisfacción se alargan. Como consecuencia, un diseño eficiente que nos dota de la capacidad de aprender en un entorno impredecible se convierte en un diseño eficiente para generar dudas existenciales en un entorno demasiado predecible, como puede ser la vida en una ciudad moderna (o, para los animales, en un zoo).

Nuestra predecible existencia no ha impedido, sin embargo, que diseñemos distintas formas de engañar a nuestro cerebro. El consumo de alcohol, nicotina, opiáceos o cannabis libera dopamina en mayor cantidad de lo habitual, generando sensación de bienestar. Otros compuestos químicos, como la cocaína, inhiben el transporte y reabsorción de la dopamina, prolongando la concentración elevada de dopamina en el espacio extracelular entre las neuronas de nuestro cerebro. Así que, en cierto modo, las adicciones de la vida moderna no son más que el comportamiento disfuncional inexorablemente ligado a un sistema que, diseñado para recibir sorpresas, es privado de ellas.

Organización jerárquica

Una de las características esenciales del *deep learning* es el diseño modular de las redes neuronales artificiales, lo que permite construir redes más complejas combinando redes más simples. Usualmente, la estructura de la red neuronal artificial se establece de antemano y el algoritmo de aprendizaje es el encargado de que la red ajuste sus pesos de la mejor forma posible. No obstante, es posible compartir módulos, incluyendo en una red parámetros específicos para una tarea concreta y parámetros compartidos para la realización de varias tareas. Las redes neuronales son capaces de aprender representaciones.¹¹⁴ Esas representaciones las podemos reutilizar, aprovechando lo ya aprendido en un contexto para otras situaciones. Es lo que se conoce en Inteligencia Artificial con el nombre de aprendizaje multitarea.

¿Existe alguna evidencia biológica que justifique ese diseño modular? Sólo tenemos que examinar el córtex cerebral con la ayuda de un microscopio. El córtex cerebral es la capa exterior de nuestro cerebro, la zona de materia gris que recubre los hemisferios cerebrales. Aunque tiene numerosos pliegues, viene a ser como una servilleta arrugada para caber en el espacio reducido del cráneo. Su estructura celular es la misma,

¹¹⁴ Jonathan Baxter. Learning internal representations. En *Proceedings of the Eighth Annual Conference on Computational Learning Theory*, COLT '95, pages 311–320, 1995. ISBN 0897917235. DOI: 10.1145/225298.225336

miremos donde miremos. El córtex se organiza en columnas con seis capas diferentes de neuronas y patrones de interconexión que se repiten, con conexiones con otras regiones del cerebro como el tálamo, conexiones inhibitorias de realimentación y conexiones excitatorias que conectan unas regiones del córtex con otras. Existe cierta variabilidad entre unas zonas y otras, si bien esa variabilidad parece deberse más a diferentes ajustes de parámetros de un mismo algoritmo que al empleo de distintos algoritmos especializados para diferentes tareas.

Así que parece que no sólo el mecanismo de aprendizaje es común (la memoria se forma fortaleciendo las neuronas que se activan conjuntamente), sino que parece que la estructura topológica de nuestra corteza cerebral responde a un patrón determinado. Su funcionamiento aún no se ha descifrado del todo, pero parece que se rige bajo unos principios generales comunes. Recordemos que si, nada más nacer, se cruzan los nervios ópticos y auditivos de un animal, el animal es capaz de ver y oír: su córtex ‘visual’ aprenderá a oír y su córtex ‘auditivo’ interpretará imágenes, una prueba de que el algoritmo utilizado por las distintas regiones del córtex es el mismo independientemente de su especialización posterior.

Un neuroanatomista español, Rafael Lorente de Nó, discípulo de Santiago Ramón y Cajal, fue el primero que se dio cuenta, en los años 20 del siglo XX, de que muchas regiones del córtex cerebral contenían el mismo tipo de unidades, que se repetían de forma iterativa a lo largo y ancho de su superficie. Son las denominadas columnas corticales.

Dado que en el córtex cerebral se realiza gran parte del procesamiento neuronal que asociamos a las capacidades cognitivas de un animal, no es extraño que su estructura se haya usado como guía para la creación de modelos de redes neuronales artificiales.

Uno de los más conocidos es el modelo de memoria temporal jerárquica de Jeff Hawkins [*HTM: Hierarchical Temporal Memory*].¹¹⁵ Hawkins intenta reproducir fielmente la estructura de seis capas del córtex cerebral humano. Su modelo, que desdeña los usos y costumbres de otros investigadores en redes neuronales artificiales que utilizan pesos reales para modelar las conexiones sinápticas, pretende ser biológicamente plausible. HTM utiliza señales y sinapsis binarias, en vez de recurrir a números con varios dígitos decimales. El algoritmo de aprendizaje de una HTM se basa en la fisiología de las neuronas piramidales del neocórtex.

Básicamente, las columnas corticales se modelan como sistemas de predicción de secuencias capaces de aprender patrones temporales de forma continua y no supervisada. Apilando múltiples columnas corticales, de forma similar a como se organizan jerárquicamente distintas regiones corticales, las HTM permiten aprender, recordar e inferir secuencias más complejas. Según Hawkins, el modelo HTM implementa la funcionalidad característica de grupos de columnas corticales relacionados jerárqui-

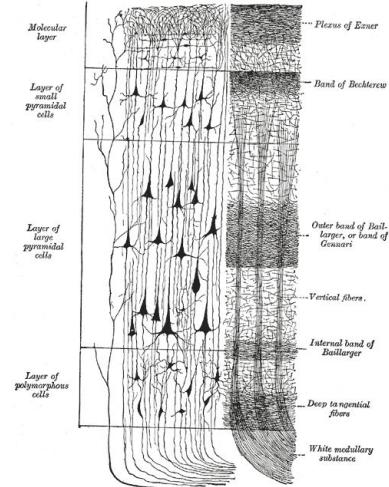


Figura 27: Las capas de neuronas del córtex cerebral. A la izquierda, grupos de células. A la derecha, sistemas de fibras. Figura 754 del libro de Anatomía de Gray, publicado en 1858 e ilustrado por Henry Vandyke Carter.

¹¹⁵ Jeff Hawkins. *On Intelligence*. Times Books, 2004. ISBN 0805074562

camente, situando el hipocampo en el nivel más alto de la jerarquía.

Como aún no se conoce con exactitud el funcionamiento preciso de las seis capas del córtex cerebral, HTM modela sólo una parte de las columnas corticales del cerebro (sus capas 2 y 3). Esto le es suficiente para detectar patrones temporales y espaciales, tratar adecuadamente el ruido en los datos de entrada y admitir cierta variabilidad en la entrada. Además, lo hace empleando una representación distribuida dispersa en la que sólo un pequeño porcentaje de las columnas corticales está activo en cada momento. Algo que también se observa en los cerebros biológicos, obligados a ello para ahorrar energía.

En redes neuronales artificiales, Geoffrey Hinton ha sugerido que deberían reemplazarse las neuronas individuales por “cápsulas” que reflejen de una forma más fiel la estructura cortical del cerebro humano.¹¹⁶ La evolución ha debido encontrar una forma eficiente de identificar características en etapas tempranas de las vías sensoriales que conducen al cerebro y reutilizar esas características en etapas posteriores. Hinton espera que esas arquitecturas, hipotéticas por el momento, sean más robustas y seguras frente a situaciones con adversarios como las que discutimos al analizar las vulnerabilidades asociadas al uso de redes neuronales artificiales. Intentar modelar regiones corticales usando una capa de neuronas en paralelo, como hacen las redes neuronales artificiales, le parece disparatado. Simplemente, era la primera forma de intentar hacerlo y resultó tener un éxito asombroso. Por desgracia, como aún no tenemos una comprensión completa del procesamiento realizado en una columna cortical, aún es pronto para saber cómo serán esas redes neuronales del futuro. La teoría favorita de Hinton es que cada columna cortical sirve para detectar concordancias entre predicciones multidimensionales.¹¹⁷ Algo muy distinto a las hipótesis binarias de Hawkins o a la combinación ponderada de señales de entrada para calcular un valor escalar que realizan las neuronas empleadas por las redes neuronales artificiales actuales. Esa teoría permitiría que las cápsulas que modelasen columnas corticales fuesen mucho más robustas frente al ruido, mejores para tratar con cambios de perspectiva y mucho mejores para segmentar (agrupando predicciones multidimensionales similares).

Las capacidades cognitivas asociadas al córtex cerebral no son las únicas para las que se ha encontrado una explicación basada en una organización jerárquica. La percepción también se ha intentado explicar en términos de una jerarquía lógica de etapas de procesamiento neuronal. Según esta interpretación, las propiedades identificadas por las neuronas visuales de una etapa determinan las propiedades de las neuronas de la siguiente etapa. Por ejemplo, algunas neuronas individuales de la zona V1 del córtex visual responden a estímulos de luz presentados en una zona relativamente pequeña del campo visual. Esa zona define los límites espaciales del campo receptivo de la neurona. Otras neuronas del córtex

El hipocampo, denominado así por el anatómista del siglo XVI Giulio Cesare Aranzio debido a su forma, parecida a la de un caballito de mar, es un componente del sistema límbico del cerebro situado en su lóbulo temporal. El hipocampo desempeña un papel fundamental en la formación de memoria a largo plazo además de, como ya mencionamos al hablar de los taxistas de Londres, estar relacionado con la memoria espacial que facilita la navegación. Brenda Milner, del Montreal Neurological Institute, observó en los años 70 que pacientes con epilepsia, a los que se les extirpaban las zonas del lóbulo temporal donde tenía su origen la actividad epiléptica, terminaban con problemas para formar nuevas memorias a largo plazo.

Brenda Milner. Clues to the cerebral organization of memory. En Pierre A. Buser y Arlette Rougeul-Buser, editores, *Cerebral Correlates of Conscious Experience*, pages 139–153. Elsevier, 1978. ISBN 0720406595

¹¹⁶ Geoffrey Hinton. AMA - Ask Me Anything. *reddit*, 2014b. URL <https://goo.gl/1dj45F>

¹¹⁷ Geoffrey Hinton. What's wrong with convolutional nets? *MIT, Brain & Cognitive Sciences - Fall Colloquium Series*, 2014a. URL <https://goo.gl/CS2Ugp>

visual responden a la presencia de gradientes en la iluminación de una escena, siendo sensibles a la orientación (gradiente espacial: barras horizontales, diagonales o verticales) o al movimiento (gradiente temporal, asociado al movimiento en una dirección concreta). Las respuestas selectivas de distintos grupos de neuronas se han estudiado con detalle para caracterizar las propiedades de sus campos receptivos. Estas propiedades pueden codificar, además de la localización, la orientación o la dirección del movimiento, la longitud de una línea, las diferencias de color, si la entrada proviene de un ojo u otro, o incluso la profundidad a la que se encuentra un objeto (aprovechando la visión estereoscópica proporcionada por las vistas ligeramente distintas de los dos ojos). En definitiva, las neuronas corticales visuales extraen características relevantes de las imágenes retinianas que nos ayudan a identificar y localizar los objetos a nuestro alrededor.

Pero no todas las neuronas que intervienen en el procesamiento visual de la información son selectivas. La retina incluye múltiples capas de neuronas especializadas de distintos tipos y el tálamo ejerce de relé de retransmisión entre la retina y el córtex (cuando estamos despiertos, conecta la señal recibida a través del nervio óptico con el córtex visual). Tanto las neuronas de la retina como las del tálamo responden a cualquier estímulo de su campo receptivo.

Los neurofisiólogos David Hubel, canadiense, y Torsten Wiesel, sueco, recibieron en 1981 el premio Nobel en Fisiología o Medicina por sus estudios sobre el procesamiento de información visual en la Universidad Johns Hopkins, que les permitieron identificar los campos receptivos selectivos de las neuronas visuales y la organización columnar del córtex visual. Sus primeros trabajos parecían confirmar que la visión involucra un análisis jerárquico de las imágenes captadas por la retina. De los valores de luminancia y su distribución espectral (colores) se extraían ángulos, longitudes, profundidad, movimiento y otras características de la imagen hasta alcanzar el nivel de abstracción que asociamos a nuestra percepción visual. Desde la retina, pasando por el nervio óptico hasta el tálamo, de ahí al córtex visual primario y, posteriormente, en otras regiones del córtex encargadas de las etapas superiores del análisis visual de una escena. De hecho, el interés original en la modularidad del córtex en los años 50 se debía al estudio del sistema visual. De ahí surgió el modelo del ‘cubito de hielo’ [*ice cube model*],¹¹⁸ un modelo de organización del córtex visual por columnas. Según ese modelo, cada pequeño segmento del córtex, al que Hubel y Wiesel denominaban “hipercolumna”, contenía un conjunto completo de elementos de procesamiento y extracción de características.

Derivada de esa interpretación estrictamente jerárquica de la percepción visual, surgió la idea de las neuronas abuela [*grandmother cells*],¹¹⁹ un término acuñado en 1969 por Jerry Lettvin, un neurocientífico del

¹¹⁸ David H. Hubel. *Eye, Brain, and Vision*. W.H. Freeman, 1988. ISBN 0716750201. URL <http://hubel.med.harvard.edu/>

¹¹⁹ Charles G. Gross. Genealogy of the “grandmother cell”. *The Neuroscientist*, 8(5):512–518, 2002. DOI: 10.1177/107385802237175

MIT. Si las características primitivas de una imagen en la retina se van procesando paulatinamente para ir extrayendo propiedades cada vez más complejas en niveles superiores del cerebro, ¿implica esto la existencia de neuronas que, en última instancia, son ridículamente selectivas? Dicho de otro modo, ¿tiene su cerebro neuronas específicas que sólo se activan si recibe en su retina una imagen de su abuela? Aunque la pregunta se planteó inicialmente de forma irónica, para ilustrar la inconsistencia lógica del concepto, muchos se la tomaron en serio. De hecho, en los años 80 se llegaron a identificar neuronas que, en efecto, responden a la presencia de caras en zonas de la memoria asociativa del cerebro del mono. Posteriormente, se comprobó que en el lóbulo temporal del cerebro humano también existe un área que responde de forma selectiva a las caras. Así que, si en su cerebro existe una neurona dedicada a la imagen de su abuela, ya sabe dónde buscarla, en su lóbulo temporal.

Percepción

Las técnicas de *deep learning* han alcanzado la fama gracias a su capacidad de imitar las capacidades perceptivas humanas, tanto a la hora de procesar sonidos en los sistemas de reconocimiento de voz como a la hora de analizar imágenes para identificar objetos en visión artificial. En realidad, su éxito se basa más en la disponibilidad de la potencia de cálculo necesaria para procesar conjuntos de entrenamiento enormes, que en simular el funcionamiento real de nuestros sistemas visual y auditivo. Pese a ello, resulta interesante analizar cómo funciona la percepción humana, tanto en sí misma, para comprender mejor el funcionamiento de nuestro sistema nervioso, como para buscar ideas que se puedan aprovechar para el diseño de redes neuronales artificiales.

Nuestro cuerpo está dotado de una amplia variedad de sensores, cuyo coste energético asociado varía notablemente. Generalmente, cuanto mayor sea la capacidad de transmisión de datos de un nervio, más grueso ha de ser su axón. Cuando doblamos el diámetro del axón, su volumen se cuadriplica. Dado que la concentración de mitocondrias es más o menos constante y nos sirve de indicador del consumo energético de la neurona, cuando se cuadriplica su volumen también lo hace su suministro energético. Nuestros sensores olfativos son lentos y no requieren un gran ancho de banda para enviar información al cerebro, por lo que sus axones son extremadamente delgados, acercándose a su límite físico. La visión requiere un mayor ancho de banda, por lo que los axones de las células ganglionares retinales que forman el nervio óptico son más gruesos. El oído es aún más rápido al transmitir señales, por lo que los axones del nervio auditivo son aún más gruesos que los del nervio óptico. Su diferencia en grosor hace que los axones auditivos más gruesos tengan un coste energético 100 veces superior a los axones olfativos.

Los sensores olfativos, igual que muchos de los que tenemos en nuestra piel, codifican directamente la señal que viaja hasta el cerebro. Sin embargo, los sensores de sonido y de movimiento de nuestra cabeza utilizan una etapa sináptica: la neurona sensorial utiliza sus vesículas sinápticas para activar un pulso eléctrico en una segunda neurona, que es la que transmite la señal hasta el cerebro a través del nervio auditivo, vestibuloclear o estatoacústico. Los fotosensores de nuestra retina utilizan dos etapas sinápticas. En una primera etapa, que funciona de forma analógica, modulan un voltaje en una segunda neurona. En una segunda etapa, se codifican pulsos que una tercera neurona envía hasta el cerebro a través del nervio óptico.

Para algunos sentidos, disponemos de *arrays* de sensores:

- En el oído de los mamíferos, una serie de células ciliadas, mecánicamente sensibles, captan las diferencias de presión asociadas al sonido. Cada célula está afinada para captar un rango particular de frecuencias. Las células se organizan a lo largo de la membrana basilar de la cóclea desde la frecuencia más baja hasta la más alta (de 20Hz a 20kHz en el caso de los seres humanos). Los axones que corresponden a las frecuencias más altas son tres veces más gruesos que los que corresponden a las más bajas, por lo que consumen unas 10 veces más volumen y energía. Teniendo esto en cuenta, no debería sorprender demasiado que la selección natural haya colocado la voz humana en la parte baja del rango de frecuencias que nuestro oído es capaz de captar (la más económica desde el punto de vista energético). Resulta curioso que el tejido cerebral, aunque caro, no sea el más caro de nuestro sistema nervioso: los nervios auditivos son los más costosos porque sus neuronas tienen restricciones de tiempo asociadas al procesamiento de frecuencias temporales elevadas.
- Desde el punto de vista espacial, un array de sensores debe ser capaz de captar los detalles que sean esenciales para realizar su tarea con éxito. En el caso de la visión humana, su resolución espacial es de 60 ciclos por grado, lo que requiere 120 conos por grado según el criterio de Nyquist. Extrapolando a dos dimensiones, esa resolución necesitaría 200,000 conos por milímetro cuadrado, lo que resultaría demasiado costoso en términos energéticos para cubrir el campo visual completo. De ahí, que se opte por la solución de tener una pequeña fracción muy densa en el array (la fóvea) y se disponga de una distribución más dispersa de fotorreceptores en el resto de la retina. La fóvea empaqueta la mitad de sus conos en una superficie que ocupa sólo el 1% de la superficie de la retina. El córtex visual, por tanto, destina la mitad de su volumen a procesar la señal recibida de la fóvea, lo que permite un análisis detallado de la parte más interesante de la imagen sin que se descontrolle el coste energético asociado al procesamiento visual.

El teorema de muestreo, formulado en forma de conjectura por Harry Nyquist en 1928 y demostrado formalmente por Claude Shannon en 1949, establece que se puede reconstruir de forma exacta una señal periódica a partir de sus muestras cuando la tasa de muestreo es superior al doble de la frecuencia más alta de la señal, i.e. $F_s > 2F_{max}$.

A cambio de ese ahorro energético, la fóvea requiere un sistema de músculos que mueva el ojo y un sistema de control para dirigir la fóvea hacia objetos de interés (los movimientos sacádicos), además de las microsacadas que mantienen al ojo en un estado de constante vibración (20 segundos de arco a 60Hz, completamente imperceptibles para nosotros). Las microsacadas son necesarias, ya que la retina sólo responde a cambios en la luminancia. Si no se produjesen, mirar fijamente a un punto haría que los estímulos enviados al cerebro desapareciesen. Los movimientos sacádicos nos permiten estabilizar un objeto en la fóvea, lo que nos permite percibirlo con una mayor resolución. Como efecto colateral, esa estabilización también reduce el rango de frecuencias temporales sobre la fóvea, lo que permite operar a las neuronas visuales a menor velocidad, con el consiguiente ahorro de energía.

- La misma estrategia que se utiliza en la retina, que contiene un área reducida mucho más sensible que el resto, también se utiliza en el tacto. Los sensores táctiles no están distribuidos de forma uniforme por toda nuestra piel, sino que se concentran en las puntas de los dedos, labios y lengua. De ahí el homúnculo distorsionado que habrá visto en algunos mapas del córtex humano.

Centrémonos ahora en el sentido más estudiado: la vista. El ojo humano le envía al cerebro información visual a una velocidad que tal vez le sorprenda: unos 10 megabits por segundo, la velocidad de una conexión Ethernet de las antiguas (ahora se usan de 100Mbps o de 1Gbps) o de una conexión ADSL barata. Más o menos, de lo que consume una señal comprimida de televisión en alta definición (HD 1080p, con 25 fotogramas por segundo de 1920x1080 píxeles) y mucho menos de lo necesario para enviar una señal 4k (Ultra HD, de 3840x2160 píxeles), ya comprimida. Las cifras indican claramente que la retina no se limita a ser un componente pasivo en el procesamiento de la información visual.

El procesamiento local de la señal visual en la retina comienza con la fototransducción en conos y bastones: un circuito proteínico amplifica químicamente la energía de un fotón y se acopla a un circuito proteínico ya eléctrico para transmitir la señal amplificada. Conos y bastones están optimizados para diferentes escenarios, por lo que cada uno de ellos tiene desventajas particulares. Sin embargo, la retina combina dos diseños especializados diferentes para sacarles el máximo partido. Un punto de luz diurna un 1% más brillante que el fondo entrega unos 10^9 fotones a un grupo de unos 4,000 conos en 100ms. Los fotones isomerizan unas 10^7 moléculas de opsina (la proteína que desencadena las transducción), reduciendo por cien el número de cuantos. Las terminales sinápticas del cono liberan unas 10^5 vesículas sinápticas (donde se alojan los neurotransmisores), reduciendo de nuevo por cien el número de cuantos de la



Figura 28: Un homúnculo sensorial, con sus miembros anatómicos en proporción a la superficie cortical dedicada a ellos. Fuente: Wikipedia Commons.

señal, que ahora es una señal analógica formada por pulsos de glutamato. Las células bipolares de la retina realizan una nueva transformación de la señal. Su respuesta tónica es prácticamente nula (de ahí la necesidad de los movimientos microsacádicos) pero cuando se activa el punto de luz generan una pequeña ráfaga de cuantos vesiculares. La presencia de apenas una decena de esos cuantos es suficiente para generar un pulso eléctrico en una célula ganglionar, que es la que transmite la señal al cerebro a través del nervio óptico. En resumen, partiendo de 10^9 fotones, se generan 10^7 eventos moleculares que acaban dando lugar a un simple pulso en uno de los axones que conectan la retina con el cerebro. Cuando se produce ese pulso, el 90 % de la información visual se ha descartado en el circuito neuronal de la retina, por lo que la retina debe procesar las imágenes para que sus células ganglionares sólo envíen lo estrictamente necesario.

La resolución espacial de la retina viene dada por su capa de salida, su array de células ganglionares. Dado que cada célula ganglionar combina las señales de 10^2 a 10^3 conos [*pooling*], la combinación de señales en las terminales de los conos no afecta a la resolución espacial del array de células ganglionares. En realidad, el acoplamiento local entre conos en la fóvea es lo suficientemente pequeño como para que el desenfoque neuronal [*neural blur*] sea inferior al desenfoque óptico [*optical blur*] debido a las propiedades mecánicas del ojo.

Desde el punto de vista de un cono, una escena no es más que una sucesión de cambios de brillo. Un claroscuro en torno a la luminancia media de la escena. Las breves variaciones de brillo son informativas, pero no la intensidad media de la señal, por lo que la retina sólo debería transmitir esas oscilaciones. Para evaluar esa media, la retina dispone de un dispositivo especializado: una capa de neuronas, las células horizontales, que expanden un árbol planar de terminaciones justo debajo de los terminales de los conos. Gracias a sus conexiones inhibitorias, la retina codifica el contraste de una imagen, no su intensidad en sí. Dado que los objetos del mundo real generan una señal óptica reflejando una fracción de la luz que recae sobre ellos, un mismo objeto envía una señal diferente en función de su iluminación. Dividiendo esa señal por una media de intensidad local, que depende de la iluminación local de la escena, se elimina parte de la variabilidad de la señal proveniente de un mismo objeto: la debida a cambios de iluminación. La señal de contraste depende más de las propiedades físicas del objeto (su reflectancia y transmitancia) que de las condiciones cambiantes de iluminación ambiental.

Las células ganglionares forman la capa de salida de la retina y reciben como entrada las señales provenientes de las células bipolares. Sus axones forman el nervio óptico. En realidad, existen múltiples tipos de células ganglionares. No se conoce su número exacto para ninguna especie, pero se sabe que en todas las retinas de mamíferos que se han

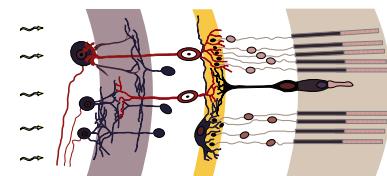


Figura 29: Las capas de neuronas de la retina humana. La luz, proveniente de la izquierda, atraviesa varias capas neuronales hasta alcanzar conos y bastones (a la derecha), donde desencadena un cambio químico que genera una señal eléctrica. La señal, en primer lugar, atraviesa células bipolares y horizontales (en el centro). A continuación, pasa por las células amacrinas y ganglionares (a la izquierda) antes de llegar al nervio óptico. Figura adaptada de un dibujo de Santiago Ramón y Cajal realizado en 1911. Fuente: Wikipedia.

En realidad, hay dos tipos de células bipolares en la retina, que funcionan a modo de rectificador. Unas codifican las zonas más claras de la imagen y otras, las más oscuras. De esa forma, una señal que puede ser positiva o negativa con respecto a la media se transmite como dos señales separadas en paralelo, ambas positivas.

Y ni siquiera hemos hablado de las células amacrinas, neuronas inhibitorias que también interactúan con las células bipolares y ganglionares...

estudiado hay, al menos, 20 tipos distintos. A diferencia del sensor de una cámara fotográfica, que transmite una imagen como una matriz perfectamente alineada de píxeles, el nervio óptico tiene una estructura que combina axones de diferentes grosores en función de la velocidad a la que deben transmitir información (para ahorrar energía, ya que su consumo crece al cuadrado de su diámetro, igual que su volumen). Algunos tipos de células ganglionares están diseñadas para detectar cambios lentos en la intensidad de la luz y se conectan directamente con el núcleo supraquiasmático [*SCN: suprachiasmatic nucleus*], el encargado de controlar los ciclos circadianos. Otro tipo, que detecta cambios breves de intensidad, se conecta directamente con el clúster de neuronas que regula el diámetro de la pupila. Otro objetivo del nervio óptico está en el cerebro medio: el colículo superior, donde se integran señales visuales, auditivas y somatosensoriales que (1) nos ayudan a decidir hacia dónde mirar, (2) activan mecanismos motores que orientan ojos, oídos y cabeza hacia ese sitio, y (3) nos permiten corregir esos movimientos instintivos. Por último, las células ganglionares también se conectan con una docena de clusters de neuronas en el tálamo, desde donde se retransmiten las señales al córtex visual primario (V1). Un diseño muy eficiente para transmitir en paralelo una veintena de señales diferentes usando unos 10^6 axones en un nervio óptico que tiene menos de dos milímetros de diámetro.

Una vez llegados al córtex visual, los circuitos de la región V1, que contiene unas 40 neuronas por cada fotorreceptor de la retina, representan las posiciones y frecuencias espaciales. Se cree que funcionan de forma similar a los filtros de Gabor. La mayor parte de las salidas del córtex visual primario van a parar a otra región del córtex conocida como V2 (engañosamente denominada, en ocasiones, córtex visual secundario). Es la última zona del sistema de percepción visual en la que una lesión causa ceguera. Más allá de V2, aún queda más de la mitad de la superficie del córtex dedicada a la visión. Por ejemplo, la franja ventral [*ventral stream*], en el lóbulo temporal inferior, contiene neuronas que responden de forma selectiva a características como color, forma o textura, útiles para identificar objetos independientemente de su orientación y posición en el campo visual. Por el contrario, la franja dorsal [*dorsal stream*], en el lóbulo temporal superior, responde selectivamente a características como profundidad, dirección y velocidad de movimiento, ideales para localizar dónde están los objetos. Esto es, distintas zonas se especializan en resolver por separado y de forma económica tareas importantes para la supervivencia de un animal. Más de medio siglo después de los estudios iniciales de Hubel y Wiesel, aún quedan muchas incógnitas y se sigue estudiando cómo funciona la percepción visual.

En resumen, la retina preprocesa la imagen para transmitir sólo lo que resulte esencial para el funcionamiento de distintos módulos del cerebro.

El córtex visual primario comienza identificando características locales de la imagen (aristas, movimiento), que luego se reutilizan y combinan de distintas formas en niveles superiores del córtex visual. Compartir esas características ahorra espacio y energía. El procesamiento local y las proyecciones ordenadas de una zona a otra también sirven para ahorrar en cableado. En las etapas finales del procesamiento visual, se separan distintos tipos de información, como el dónde (franja dorsal) del qué (franja ventral).

Los estudios de Hubel y Wiesel sobre el procesamiento de la información visual en el cerebro dieron lugar a un modelo jerárquico lógico, atractivo por su elegancia pero no del todo correcto. Con el paso del tiempo, se fueron acumulando evidencias acerca del funcionamiento de distintos tipos de neuronas que intervienen en la percepción visual, su conectividad y la organización general del sistema visual. La idea de un sistema estrictamente jerárquico, en general, y del modelo del cubito de hielo, en particular, no se sostenía. Era como intentar resolver la cuadratura del círculo: *“a square peg... into a round hole”* en palabras del neurocientífico Dale Purves, de la Universidad de Duke.

Desde los años 80, se empezó a dudar de que las respuestas neuronales a los estímulos visuales, incluso a los más simples, se pudiesen racionalizar en términos de una jerarquía que comienza con la detección de características al nivel de la imagen percibida por la retina y termina con una representación abstracta del contenido de la imagen en el córtex cerebral. Por un lado, diferentes estímulos pueden desencadenar los mismos patrones de actividad cortical visual. Por otro lado, los mismos estímulos pueden percibirse de forma radicalmente diferente en función del contexto. Según Purves, analizar detalladamente el funcionamiento de los distintos tipos de neuronas nunca nos permitiría comprender realmente la percepción humana ni sus mecanismos subyacentes.

Las limitaciones de cualquier aproximación lógica a la percepción visual ya aparecían en el *Ensayo hacia una nueva teoría de la visión*, publicado por el filósofo irlandés George Berkeley en 1709. La misma imagen en la retina se puede generar de muchas formas diferentes, por objetos de diferentes tamaños, a distintas distancias del observador y en diferentes orientaciones. Es el problema óptico inverso, que nos obliga a reconocer que la fuente real de una imagen es inevitablemente incierta. No se trata simplemente de que haya imágenes que puedan ser ambiguas, es que resulta imposible determinar con exactitud los objetos tridimensionales que proyectan una imagen bidimensional sobre nuestras retinas.

Nuestro sistema nervioso, esencialmente, se dedica a una sola cosa: enlazar información sensorial (percepciones o sus equivalentes subconscientes) con comportamientos adecuados. Desde ese punto de vista, no es más que un sistema que genera una salida (comportamiento) a partir de una entrada (percepción). Internamente, la conexión entre percepción

Otros sentidos, como el olfato y el gusto, no requieren un procesamiento tan elaborado. Entre otras cosas, porque no disponen de características locales. Las moléculas que indican un olor llegan a través del aire, pero no existen correlaciones espaciales que nos permitan identificar su origen.

y comportamiento se establece configurando las conexiones sinápticas que existen entre unas neuronas y otras, algo que se consigue mediante un mecanismo de prueba y error. El problema óptico inverso y los problemas equivalentes para otras modalidades sensoriales no dan muchas más opciones.

La complejidad del funcionamiento del cerebro se reduce a utilizar, crear y modificar conexiones neuronales. Aprovechando, obviamente, la estructura de la que la evolución lo ha dotado por medio de la selección natural. ¿Por qué este enfoque tan reduccionista? Porque, según Purves, existen demasiadas peculiaridades de la experiencia subjetiva que sólo pueden explicarse de esta forma. Las discrepancias universales que se observan entre la percepción humana y las medidas meramente físicas no serían anomalías, a las que denominamos “ilusiones” cuando resultan más flagrantes. Tampoco serían el resultado de las limitaciones de nuestros circuitos neuronales. Simplemente, son la firma dejada por el funcionamiento del cerebro. De ahí que Dale Purves proponga el estudio de las “ilusiones” perceptuales, no como anomalías, sino como marcas características de la forma en la que el cerebro humano funciona realmente.¹²⁰

Relajémonos un poco disfrutando de algunas de las peculiaridades de nuestra percepción de la luminancia, el color, la geometría y el movimiento:

■ *Luminancia*

La forma en la que las superficies reflejan una misma cantidad de luz puede percibirse de manera muy diferente. La causa de estos efectos visuales puede provenir de tener que resolver el problema óptico inverso.

Puede existir una discrepancia entre la luminancia de una superficie y la percepción de la misma. Si usamos un fotómetro (u ocultamos su contexto), podemos comprobar que los círculos centrales reflejan la misma cantidad de luz. Sin embargo, el círculo sobre un fondo oscuro parece más claro que el círculo sobre un fondo claro.

Si los valores de brillo que percibimos fuesen simplemente proporcionales a los valores de luminancia, no nos serían de demasiada utilidad para guiar nuestro comportamiento. Sin embargo, si esa percepción proviene de un proceso empírico de prueba y error, la evidencia se acumularía a favor de los patrones de luz que provienen de escenas naturales y permiten determinar el comportamiento adecuado. No se resuelve realmente el problema óptico inverso, sino que, simplemente, se esquiva.

En los años 50, se explicaban esas discrepancias entre la percepción del brillo y la luminancia porque se sabía que el campo receptivo visual de

¹²⁰ Dale Purves. *Brains: How they seem to work and what that tells us about who we are*. FT Press Science, Pearson, 2010. ISBN 0137055099

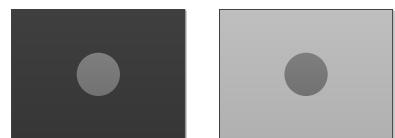


Figura 30: Luminancia: Discrepancia en la percepción debida al contexto.

una célula de salida de la retina estaba rodeado de una región con la polaridad invertida de inhibición lateral, organización que presumiblemente ayudaba a detectar fronteras. Esa observación electrofisiológica permitía suponer que el círculo sobre un fondo oscuro se veía más claro que el círculo sobre un fondo claro por las características específicas del campo receptivo local de cada neurona de la retina.

Sin embargo, existen otros patrones para los que la explicación basada en los campos receptivos locales no resulta satisfactoria. En la figura, las dos franjas grises superiores están rodeadas de tres zonas más claras que las dos zonas oscuras que acompañan a las tres zonas grises de debajo. Aun siendo del mismo tono de gris, parece que las dos franjas superiores son más claras que las tres inferiores. Pese a ser adyacentes a zonas más claras, nuestra percepción nos engaña en el sentido contrario que en el ejemplo anterior.

El contexto, pues, influye en nuestra percepción del brillo, pero no siempre en el mismo sentido. La diferencia de brillo aparente es similar en ambos casos pero en el primero se ve más claro cuando el círculo gris está rodeado de zonas más oscuras y en el segundo se ven más claras las dos franjas grises de la cara superior del objeto (que nuestro cerebro interpreta como la cara iluminada de esa escena falsamente tridimensional).

La experiencia acumulada en la interpretación de escenas naturales es lo que hace que, para los mismos valores de luminancia, tendamos a interpretar como más brillante un objeto cuando se encuentra en un entorno oscuro que cuando se encuentra en un entorno más claro. Es una explicación plausible que no requiere siquiera analizar las propiedades fisiológicas de las neuronas que intervienen en la percepción visual.

El efecto White es otro caso en el que se produce un efecto visual similar. En este caso, algunos fragmentos de las franjas blancas y negras se reemplazan por un rectángulo gris siempre del mismo tono. Las barras verticales resultantes, aunque son del mismo color y opacidad, se perciben de forma claramente diferente. Los rectángulos de la izquierda se perciben como más brillantes pese a estar rodeados de más zonas blancas que negras. Los de la derecha, parecen más oscuros aunque estén más rodeados por zonas negras. La inhibición lateral, que explicaba la primera ilusión de esta sección, no puede explicar este fenómeno. Un efecto similar, conocido como ilusión de Munker, ocurre cuando las franjas son de colores diferentes.

Las bandas de Mach provienen de una observación de 1865 del físico alemán Ernst Mach, cuyo nombre le sonará por utilizarse como unidad de medida asociada a la velocidad del sonido. Cuando se produce un gradiente de luminancia, de una zona más clara a una zona más

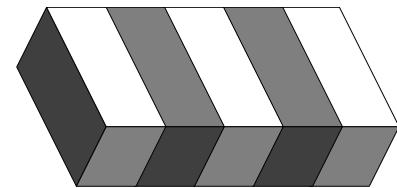


Figura 31: Luminancia en 3D: Discrepancia en la percepción debida a la interpretación tridimensional de la imagen.



Figura 32: Luminancia: El efecto White. Este ejemplo sirve para descartar las explicaciones de las diferencias en la percepción del brillo basadas en las propiedades del campo receptivo local de las neuronas que procesan la señal visual.

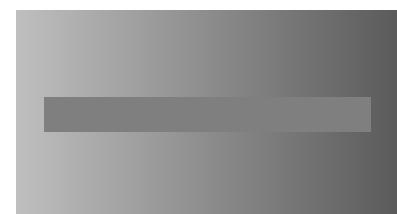


Figura 33: Luminancia: Ilusión de contraste simultáneo. El fondo es un gradiente, de gris claro a gris oscuro. El rectángulo central es de un color sólido constante, aunque parece tener un gradiente inverso al del fondo.



Figura 34: Luminancia: Bandas de Mach. El comienzo y el final del gradiente parecen incluir bandas de mayor y menor brillo, respectivamente.

oscura, nuestra percepción nos hace creer que existe una banda de mayor brillo al comienzo del gradiente y otra de menor brillo al final del mismo. Sin embargo, estas bandas no existen físicamente, ya que el gradiente no es más que una transición gradual uniforme de un nivel de luminancia a otro.

Otro efecto similar que involucra gradientes es el efecto de la frontera de Cornsweet. El psicólogo Tom Cornsweet describió este efecto un siglo después de la observación de Mach. En este caso, un gradiente del gris al negro opuesto a un gradiente del gris al blanco hace que niveles idénticos de luminancia parezcan diferentes. La superficie correspondiente al gradiente del gris al negro parece más oscura que la superficie adyacente con el gradiente del gris al blanco. Además, se aprecian claramente las bandas de Mach.

■ *Color*

Como sucedía con la luminancia, nuestra percepción del color tampoco corresponde a las medidas que proporciona un espectrofotómetro. La escena entera es relevante para la percepción del color de cualquier parte de ella.

La interpretación de los estímulos que elicitan nuestra percepción del color, como sucede con el brillo, dependen de nuestra experiencia previa. Sin datos acerca de esa experiencia en la interpretación de escenas naturales en color, nunca podremos comprender nuestra visión tricromática. Nuestra percepción del color no es más que el resultado de una estrategia empírica que nos permite lidiar con el problema óptico inverso.

Dado que la percepción del color involucra cualidades como tono [*hue*], saturación [*saturation*] y brillo [*brightness* o *lightness*], las pruebas colorimétricas se diseñan para analizar cómo interactúan esas cualidades. Por ejemplo, se ha observado que cambios en el tono de un color afectan a la percepción del brillo y que cambios en el brillo afectan a la percepción del tono de un color.

En pruebas de emparejamiento de colores, se ha mostrado que la percepción de la saturación del color varía en función de la luminancia (efecto Hunt), que la percepción del tono cambia en función de la saturación (efecto Abney), que la percepción del tono se altera en función de la luminancia (efecto Bezold–Brücke) y que la percepción del brillo se distorsiona con cambios en el tono y la saturación (el efecto Helmholtz–Kohlrausch).

En pruebas de discriminación de colores, la capacidad de distinguir diferencias en tono, saturación o brillo también cambia dependiendo de la longitud de onda asociada al estímulo recibido.

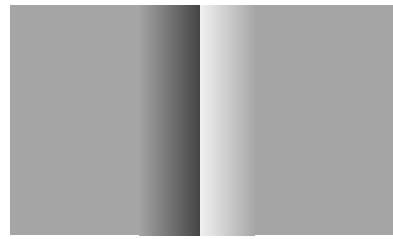


Figura 35: Luminancia: Frontera de Cornsweet. Las superficies a izquierda y derecha son del mismo color, aunque no lo parezca.

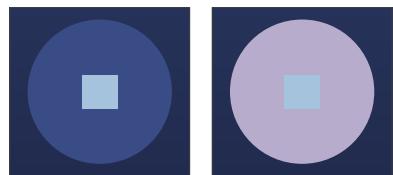


Figura 36: Percepción del color en una típica prueba de contraste. Aunque los cuadrados centrales son del mismo color al medir su espectro, se perciben como si fuesen de tonos ligeramente distintos.

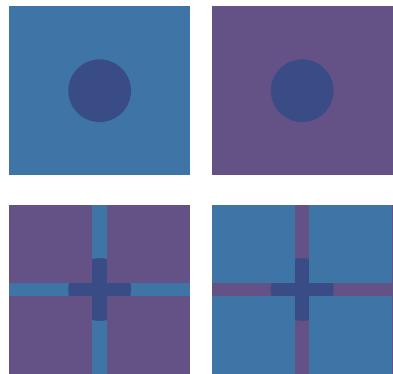
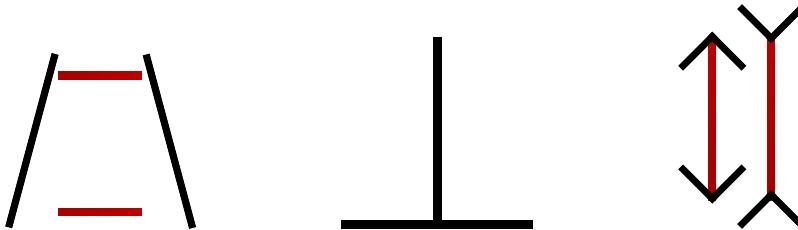


Figura 37: Percepción del color. Arriba, una prueba de contraste: dos círculos del mismo color parecen diferentes al estar rodeados de regiones de distinto color. Debajo, una prueba de constancia: los círculos centrales siguen siendo los mismos, parcialmente ocultados por regiones en las que los colores dominantes se han intercambiado. Pese a ello, los colores se siguen percibiendo como en la fila de arriba.



Las diferencias entre lo que percibimos y la realidad física que observamos nos suelen pasar inadvertidas. Nuestro éxito al desenvolversemos en un mundo que sólo percibimos de forma indirecta nos permite no ser conscientes de las diferencias sistemáticas que existen entre lo que percibimos y lo que realmente está ahí. No vemos el mundo tal como es, sino que todas nuestras percepciones son, en realidad, ilusiones. Nuestro sistema visual lo hace tan bien, que casi todo el mundo diría que está viendo la realidad. Se ignora que cada uno ve el mundo según el color del cristal con que mira, como reza el famoso poema de Ramón de Campoamor: “En este mundo traidor / nada es verdad ni mentira / todo es según el color / del cristal con que se mira”.

■ Geometría

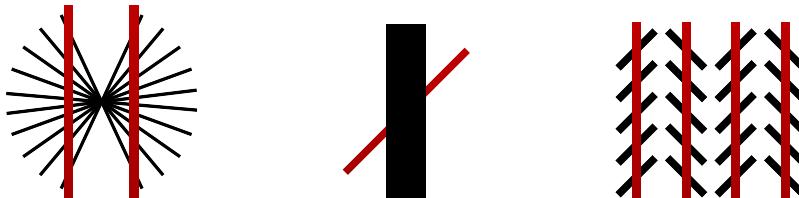
Como sucede con la percepción del color, las cualidades visuales que nos permiten realizar comparaciones para identificar similitudes y diferencias entre objetos físicos también se perciben de forma subjetiva. Nuestro sistema visual lo hace tan bien, que llegamos a pensar que la geometría que percibimos es exactamente la geometría de los objetos en el mundo real. Esa creencia nos hace percibir, como ilusiones, efectos que no son más que la prueba de cómo funciona nuestro sistema visual a la hora de resolver el problema óptico inverso cuando interpretamos las propiedades geométricas de los objetos en una escena.

Existen discrepancias entre la longitud que podemos medir con ayuda de un metro o una regla y la longitud que percibimos:

- Nuestra costumbre a la hora de interpretar escenas del mundo natural hace que, aunque dos líneas tengan exactamente la misma longitud, su contexto puede hacernos creer que se trata de una representación bidimensional de una escena tridimensional. La línea aparentemente más lejana parece ser más larga que la aparentemente cercana, cuando no es así.

Es, en el fondo, el mismo motivo por el que la Luna asomando por el horizonte nos parece mucho más grande de lo que es en realidad. Algo que sólo descubrimos al comprobar con decepción que la espectacular foto que creímos haber tomado es, en realidad, igual que cualquier otra foto que hubiésemos tomado con la Luna

Figura 38: Percepción geométrica de la longitud: Una interpretación tridimensional de una escena bidimensional nos inclina a pensar que la línea horizontal más ‘lejana’ es más larga que la ‘cercaña’ (izquierda). La línea vertical parece más larga que la horizontal (centro). La ilusión de Müller-Lyer nos hace creer que las dos flechas no son de la misma longitud, cuando en realidad sí lo son (derecha).



en otra posición con respecto al horizonte. El ángulo subtendido por el diámetro de la luna vista desde la Tierra es siempre de apenas medio grado. Más o menos el mismo ángulo que el del Sol, que es mucho más grande pero también está mucho más lejos, por muy atractivo que encontremos su puesta de Sol (aunque, en este caso, posiblemente sea la percepción del color la que nos resulte más llamativa).

- Una línea parece más larga cuando se presenta verticalmente que cuando se muestra horizontalmente, como descubrió el psicólogo alemán Wilhelm Wundt en 1858. Para que parezcan de la misma longitud, la línea horizontal tendría que ser un 30 % más larga que la vertical. Sin duda, este efecto se puede explicar por las escenas a las que estamos acostumbrados en el mundo natural, en el que predominan las líneas horizontales como la del horizonte. Sin embargo, seguramente le sorprenderá descubrir que la longitud máxima de la línea se percibe cuando tiene una inclinación de 30° sobre la vertical.
- La ilusión de Müller-Lyer consiste en visualizar una flecha estilizada. Cuando se le pide a alguien que marque su punto medio, invariablemente escogerá un punto más cerca de la pluma de la flecha que de su punta. Cuando se cambia la representación de los extremos de la flecha, se modifica nuestra percepción visual de la longitud de la flecha. Aunque dos flechas tengan la misma longitud, nos parece más corta la flecha con dos puntas que la flecha con dos plumas.

Estas discrepancias entre la longitud real de algo y su percepción visual desafían cualquier explicación meramente lógica. De la misma forma, también se observan peculiaridades en nuestra percepción del paralelismo. Veamos tres ejemplos descubiertos en el siglo XIX:

- La ilusión de Hering, descubierta por el fisiólogo alemán Ewald Hering: Cuando de forma instintiva dotamos de profundidad a una imagen bidimensional, líneas que son en realidad paralelas dejan de parecerlo. Las ilusiones de Wundt, inversa de la de Hering, y de Orbison, algo más compleja, también muestran cómo se distorsiona nuestra percepción visual de las figuras geométricas dependiendo

Figura 39: Percepción geométrica del paralelismo. Ilusión de Hering: Las dos líneas paralelas se combinan al aparecer junto a una serie de líneas que dota de profundidad a la imagen bidimensional (izquierda). Ilusión de Poggendorff: Una única línea recta, ocluida parcialmente, da la sensación de que no es única, sino que está formada por dos segmentos paralelos pero no alineados (centro). Ilusión de Zöllner, en la que líneas paralelas parecen inclinadas por el contexto en el que se muestran (derecha).

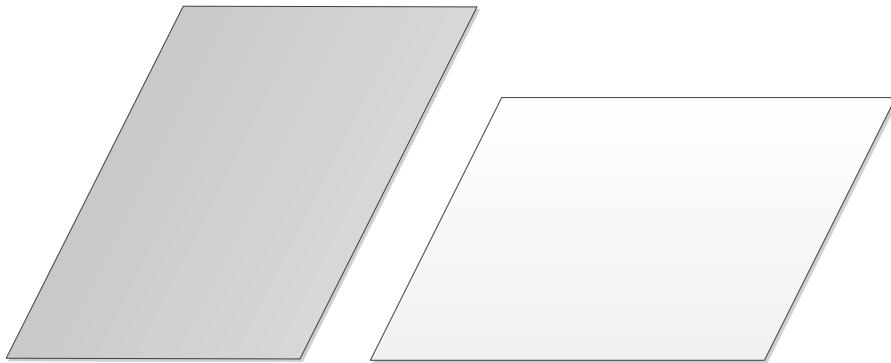


Figura 40: Percepción geométrica: La superficie de ambos paralelogramos es exactamente la misma.

del contexto en el que se muestran.

- La ilusión de Poggendorff, descubierta por el físico alemán Johann Christian Poggendorff: Cuando una línea recta aparece ocluida por un objeto que nos impide ver un segmento de la misma, la línea parece estar partida. Observamos dos segmentos paralelos que no parecen estar realmente alineados. El efecto se debe, al menos parcialmente, a que tendemos a amplificar los ángulos agudos (el efecto desaparecería si la línea partida fuese horizontal).
- La ilusión óptica de Zöllner, descubierta por el astrofísico alemán Johann Karl Friedrich Zöllner: Varias líneas paralelas se interrumpen con líneas paralelas más cortas, inclinadas con respecto a las primeras. El ángulo de las líneas cortas hace que veamos inclinadas las líneas más largas.

No sólo percibimos de forma errónea longitudes y ángulos, sino que nuestra percepción de las superficies también resulta engañosa. En la figura, los dos paralelogramos tienen exactamente la misma superficie. A pesar de ello, no nos lo parece. Nuestra experiencia en la interpretación de escenas naturales nos hace pensar que uno de los paralelogramos tiene mayor superficie que el otro.

■ *Movimiento*

Como sucedía con la luminancia, el color o la geometría, no podemos fiarnos de que nuestros sentidos nos proporcionen una representación fiel del movimiento. La velocidad y la dirección son los dos ingredientes básicos de la percepción del movimiento.

- La percepción de la velocidad influye en nuestra percepción de la posición de los objetos de una escena. Cuando un objeto se mueve a velocidad constante y se muestra un segundo objeto de forma momentánea, a la altura del objeto en movimiento, el flash de ese segundo objeto parece aparecer con retraso con respecto al objeto

en movimiento. Cuanto mayor es la velocidad del estímulo, mayor parece el retraso del flash (efecto *flash-lag*).

Cuando el flash aparece al comienzo de la trayectoria del objeto, parece desplazado en la dirección del movimiento (efecto Fröhlich). Cuando se le pide al observador que indique la posición del flash en presencia de un movimiento cercano, la posición percibida se desplaza sistemáticamente en la dirección del movimiento (efecto *flash-drag*).

Uno no se da cuenta normalmente de esas discrepancias, pero son reales y revelan diferencias entre la velocidad proyectada en la retina y la velocidad que percibimos. Las explicaciones tradicionales recurren a asociar esas discrepancias sistemáticas con etapas del análisis de una secuencia de imágenes sobre la retina, desde el punto de vista temporal o desde el punto de vista espacial. La teoría temporal asigna la discrepancia observada a la compensación que realiza el sistema visual para ajustar los retardos de procesamiento neuronal: el tiempo aparente en el que se muestra el flash se ajusta con respecto a los retardos de procesamiento anticipados (los axones neuronales conducen las señales eléctricas con cierta lentitud). La teoría espacial sugiere que la visión utiliza el movimiento observado para predecir la posición de los objetos móviles, empujando sus posiciones aparentes a lo largo de su trayectoria. Una explicación alternativa propone que las cualidades perceptuales del movimiento percibido se organicen en el espacio perceptual de acuerdo a la frecuencia asociada a las velocidades proyectadas en la retina.

- Nuestra percepción de la dirección del movimiento tampoco es del todo fiable, como muestran los efectos que se observan en aperturas parcialmente ocluidas, como el protector de la lente de una cámara de fotos compacta. Cuando un objeto móvil se ve a través de una apertura que ocluye parcialmente su movimiento, la dirección del movimiento parece cambiar de dirección.

Por ejemplo, si una barra inclinada 45° se mueve horizontalmente de izquierda a derecha con una velocidad constante y la observamos a través de una apertura circular que ocluye los extremos de la barra, el movimiento parece estar realizándose diagonalmente hacia abajo, con una inclinación de 45° con respecto al eje horizontal del movimiento real de la barra.

La explicación más popular de este fenómeno asume que el sistema visual calcula la velocidad local de cada objeto en una secuencia de imágenes (el denominado flujo óptico). La dirección ambigua del movimiento visto a través de la apertura se resuelve usando conocimiento a priori acerca de cómo se mueven normalmente los objetos en el mundo tridimensional. Sin embargo, esta explicación

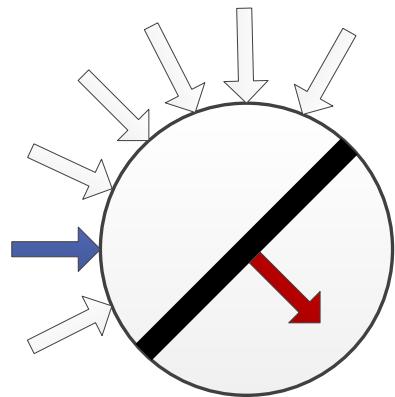


Figura 41: Percepción del movimiento: Aunque un objeto se esté moviendo horizontalmente, como sólo vemos parcialmente el movimiento a través de una apertura circular, nuestro sistema visual interpreta que el movimiento se produce en la dirección perpendicular a la inclinación del objeto. Un fenómeno que habrá observado si dispone de una cámara de fotos compacta.

es incapaz de explicar las direcciones observadas en distintos tipos de aperturas (p.ej. aperturas verticales).

Fenómenos tradicionalmente considerados como anomalías, como el *flash-lag* o los efectos de apertura, se pueden intentar explicar si consideramos que nuestra percepción del movimiento tiene una base empírica. Normalmente funciona tan bien, que nos resulta difícil creer que el movimiento observado no corresponde al movimiento de un objeto real, como los que nos encontramos habitualmente en la naturaleza. De ahí que nuestra percepción nos pueda llevar a engaños.

Los esfuerzos en comprender nuestra percepción, desde un punto de vista científico, datan del siglo XIX, cuando el físico y filósofo Gustav Fechner decidió estudiar la conexión entre lo que él llamó los “mundos físicos y psicológicos”. De ahí proviene el término psicofísica [*psychophysics*].

En el sistema visual, el objetivo de la psicofísica es comprender cómo se perciben cualidades como el brillo, el color, la forma o el movimiento. La forma que tiene el cerebro de organizar esas percepciones se denomina, a falta de un término más ingenioso, espacio perceptual.

Las distintas cualidades percibidas interactúan entre sí, como hemos visto en el caso del tono, la saturación y el brillo de un color. No sólo eso, sino que también existen interacciones entre distintas modalidades sensoriales. Esas interacciones, normalmente, nos ayudan a desenvolvernos mejor en nuestra vida diaria. Sin embargo, pueden ocasionar errores en nuestra percepción, incorrectamente etiquetados como ilusiones habitualmente. Lo que oímos puede afectar a lo que vemos. Lo que vemos, a lo que oímos. Igual que cuando uno intenta leer los nombres de una serie de colores, con cada nombre coloreado con un color diferente al color denotado por el nombre. La interacción entre la percepción visual del color y el significado que le atribuimos a las palabras que leemos hace que tengamos serias dificultades para leer las palabras tal como aparecen escritas.

El cerebro humano es tan complejo que explota todas las posibles interacciones. De esa forma, le saca el máximo partido posible a la información que nos llega a través de los sentidos. Si no lo hiciese, no estaría haciendo bien su trabajo.

Cuando intentamos reproducir el funcionamiento de una red neuronal biológica en el diseño de redes neuronales artificiales, podemos aprovechar muchas de las ideas y descubrimientos de la neurociencia. Por ejemplo, igual que se hace en las etapas iniciales del sistema visual, durante el procesamiento local de una imagen en la retina y en el córtex visual primario, podemos construir módulos genéricos, que luego reutilizaremos para tareas específicas. Es lo que se hace en el aprendizaje multitarea que mencionamos al analizar la organización jerárquica de las redes

neuronales modulares utilizadas en *deep learning*.

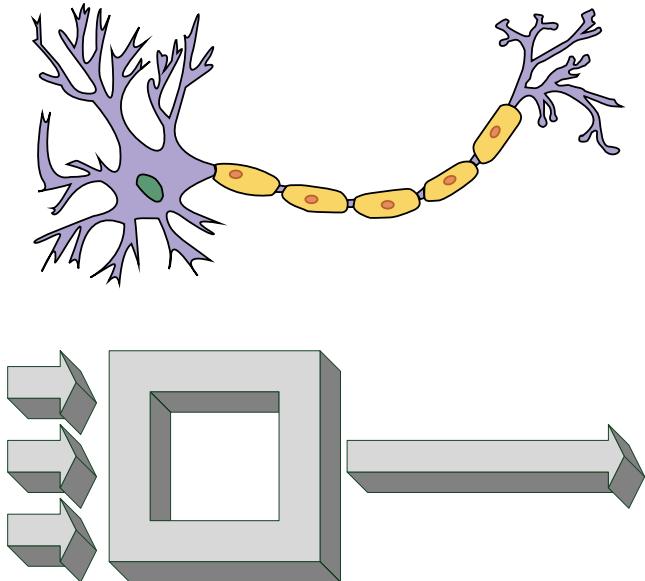
Como sucede con el cerebro humano, compartir módulos en los que se usen los mismos parámetros para resolver varias tareas puede dar lugar a la aparición de “ilusiones”. Utilizar esos parámetros comunes en combinación con parámetros específicos para tareas específicas puede dotar a las redes neuronales artificiales de mayor capacidad de generalización. Pero esa capacidad, como todo en la vida, tiene su precio. Esas redes neuronales artificiales pueden incorporar errores sistemáticos de percepción, también conocidos como sesgos, que no siempre seremos capaces de diagnosticar, identificar y corregir adecuadamente.

Modelos de neuronas y redes neuronales artificiales

Un modelo es siempre una simplificación de la realidad. Por tanto, nunca será del todo correcto. Pese a ello, algunos modelos nos pueden resultar útiles en la práctica.

En palabras de Manfred Eigen, Premio Nobel de Química, una teoría sólo tiene la alternativa de ser correcta o equivocada. Un modelo incluye una tercera posibilidad, que sea correcto pero irrelevante.

El principal riesgo que corremos al intentar modelar nuestro cerebro al nivel de neuronas individuales es que nuestro modelo puede ser tan ineficiente como intentar descubrir los principios del vuelo analizando la microbiología de los pájaros. Por ello, resulta mucho más interesante analizar el comportamiento de las redes neuronales a nivel de estructuras neuroanatómicas. Es la perspectiva modular característica del deep learning.



En función de qué aspectos deseemos analizar del comportamiento de una red neuronal de tipo biológico, las características que deberían estar presentes en nuestro modelo deberían cambiar. Desde el punto de vista computacional, resultaría prohibitivo intentar simular una red neuronal compleja a nivel molecular, si bien pueden existir situaciones en las que sean precisamente esos detalles moleculares los que nos interese analizar. Cualquier modelo computacional que creemos de una neurona puede tener su interés desde el punto de vista de la neurociencia computacional,

Figura 42: Modelo estilizado de una neurona. Arriba: Dibujo esquemático de las partes de una neurona biológica (fuente: Wikipedia Commons). Abajo: Modelo abstracto de una neurona artificial. La neurona recibe una serie de señales de entrada a través de dendritas que la conectan a otras neuronas mediante sinapsis excitatorias e inhibitorias. El cuerpo o soma de la neurona combina e integra esas señales recibidas. En función de las circunstancias, la neurona es capaz de generar un pulso eléctrico [spike] de salida que se transmite a lo largo de su axón.

que intenta descubrir las bases computacionales del funcionamiento de nuestro sistema nervioso. Ahora bien, desde el punto de vista de la Inteligencia Artificial en general y del *deep learning* en particular, nos interesará construir modelos computacionales cuya simulación en un ordenador sea lo más eficiente posible.

Salvo que estemos diseñando interfaces cerebro-ordenador [*BCI: Brain-Computer Interfaces*], posiblemente no nos interesará demasiado analizar con detalle el funcionamiento de los canales de iones individuales en la membrana de la neurona. Ni siquiera la compleja dinámica asociada a la liberación y reabsorción de neurotransmisores en el espacio sináptico entre dos neuronas. Tampoco, posiblemente, las variaciones transitorias de potencial eléctrico que se producen cuando se desencadena la reacción que da lugar a un pulso neuronal [*spike*], de sólo un centenar de milivoltios.

Ni siquiera es necesario que nuestro modelo sea biológicamente plausible cuando intentamos construir sistemas neuromórficos. Los sistemas que se emplean en la computación neuromórfica son sistemas computacionales inspirados en el sistema nervioso, que se pueden diseñar utilizando hardware de propósito específico o simulaciones software sobre hardware convencional. Obviamente, si utilizamos dispositivos microelectrónicos, sus fundamentos físicos no son los mismos que los de una célula biológica, por lo que, en vez de hablar de bombas de iones o de la liberación de vesículas presinápticas, usaremos modelos adecuados para los dispositivos físicos que estemos empleando para emular el funcionamiento de una red neuronal.

Desde el punto de vista computacional, puede que lo único que nos interese es que la señal proveniente de una entrada afecta al estado de una neurona. Amplificada o atenuada de diferentes formas, se combina con otras señales de entrada para determinar el nivel de activación de la neurona. En función de ese nivel de activación, la salida de la neurona podrá variar (o no). De ahí que, habitualmente, se modelen las sinapsis como un simple número real, conocido normalmente como peso asociado a la conexión.

Aunque esa simplificación del funcionamiento de una neurona pueda parecer extrema, no lo es si capta para nosotros los detalles realmente relevantes del funcionamiento del cerebro: la transmisión de señales de un punto a otro y la plasticidad de las conexiones neuronales, que podemos modelar variando los pesos asociados a ellas. El modelo resultante no será siempre fiel a la biología. De hecho, muchas veces no será biológicamente plausible. Aun así, puede que siga teniendo interés a la hora de resolver problemas prácticos, desde el punto de vista de un ingeniero más que desde el de un neurocientífico.

Neuronas o elementos de procesamiento

Existen diferentes aspectos de una neurona biológica que podemos descartar cuando construimos un modelo computacional con el que construir una red neuronal artificial:¹²¹

- *Abstracción espacial*

Aunque las neuronas biológicas ocupan un espacio tridimensional en el cerebro de un animal, lo que les impone restricciones con respecto a su configuración e interconectividad, los modelos de neuronas artificiales que utilicemos no han de respetar las dimensiones físicas de una neurona real. Si queremos implementarlas en hardware, las técnicas de fabricación de dispositivos electrónicos que utilicemos posiblemente nos impondrán sus restricciones, en ocasiones más restrictivas que las existentes en el interior de un cerebro biológico. No en vano, la tecnología comercial de fabricación de circuitos integrados es, básicamente bidimensional.

Desde el punto de vista computacional, podemos considerar cada neurona como un nodo. Las conexiones entre unas neuronas y otras las podemos modelar mediante enlaces. La efectividad de una sinapsis a la hora de desencadenar una reacción en la neurona postsináptica depende de distintos factores, como su localización espacial (proximal cuando está cerca del soma de la neurona, distal cuando está lejos) o su propia eficiencia (dependiendo de la densidad y tipos de vesículas presinápticas y receptores postsinápticos). Por ejemplo, una sinapsis inhibitoria cerca del cuerpo de la célula postsináptica puede ser mucho más efectiva para inhibir la activación de la neurona postsináptica que la misma sinapsis en otra posición más distal en el árbol dendrítico de la neurona. Afortunadamente, todas esas variaciones las podemos modelar fácilmente con ayuda de pesos numéricos. Esos pesos serán positivos para sinapsis excitatorias y negativos para sinapsis inhibitorias. El resultado: una red formada por nodos que representan neuronas y enlaces que representan sinapsis. Los enlaces, con pesos, dan lugar a un grafo ponderado, generalmente dirigido, aunque en algunos modelos de redes neuronales también se admiten enlaces no dirigidos o, incluso, enlaces no simétricos.

- *Abstracción temporal*

En una neurona biológica, la transmisión de una señal a lo largo del axón lleva asociado un retardo que depende de la longitud del axón, su grosor y de si está mielinado o no. El retardo de propagación de la señal a lo largo de un axón, generalmente, no lo tendremos en cuenta en los modelos computacionales de redes neuronales.

La dinámica de funcionamiento de una neurona biológica involucra

¹²¹ Keith L. Downing. *Intelligence Emerging: Adaptivity and Search in Evolving Neural Systems*. MIT Press, 2015. ISBN 0262029138

La mielina [*myelin*] es una sustancia grasa que rodea los axones de determinados tipos de neuronas. Funciona como aislante alrededor del “cable” del axón y permite que los pulsos se transmitan más rápidamente. Los axones mielinizados tienen un aspecto blanquecino, de ahí el nombre que reciben las zonas del cerebro en las que predominan: materia blanca frente a las materia gris del córtex cerebral, el cerebelo y algunas estructuras subcorticales. Curiosamente, la materia gris no es gris, sino más bien rosada, por los capilares sanguíneos que aportan nutrientes a las neuronas.

un complejo mecanismo de formación de un potencial eléctrico en la membrana de la neurona, la interacción de distintos tipos de canales de iones durante la formación y propagación de un pulso, y un período de refracción durante el cual la neurona no puede volver a generar otro pulso. Todo ello se produce a una escala de milisegundos, mucho más lenta que el funcionamiento de un circuito microelectrónico que puede operar en nanosegundos (un millón de veces más rápido que una neurona biológica). Normalmente, prescindiremos de todos esos detalles de temporización y discretizaremos la componente temporal de nuestros modelos neuronales, de la misma forma que la mayor parte de los sistemas de simulación se basan en la simulación de eventos discretos, incluso aunque se empleen para modelar sistemas continuos.

Salvo, claro está, que estemos estudiando de forma fiel el funcionamiento de redes neuronales biológicas, ya sea para analizar su dinámica o para interactuar con ellas (p.ej. BCI). Existen múltiples modelos que intentan reproducir la forma en la que el cerebro codifica los datos. A menudo, parece estar relacionada con la mera presencia de pulsos. A veces, se vincula a la temporización de distintos pulsos que interactúan. En ocasiones, se relaciona con la frecuencia de activación de la neurona (pulsos o *spikes* por segundo). Existen múltiples *spiking models* en la literatura de neurociencia computacional,¹²² como el SRM [*Spike Response Model*], el modelo de Izhikevich o los modelos CTRNN [*continuous-time recurrent neural network*].

■ Abstracción funcional

Los modelos computacionales que se utilizan en Inteligencia Artificial suelen prescindir de detalles espaciotemporales que sí son relevantes para la neurociencia. No obstante, si queremos construir un modelo de neurona biológicamente plausible, el punto de partida habitual es el modelo de Hodgkin y Huxley.

Los fisiólogos y biofísicos ingleses, Alan Lloyd Hodgkin y Andrew Fielding Huxley, describieron en 1952 los mecanismos iónicos por los que se producían y propagaban los pulsos a lo largo del axón gigante del calamar.¹²³ Por este trabajo recibirían el premio Nobel de Fisiología o Medicina unos años después, en 1963. ¿Por qué un calamar?, se estará preguntando. Tal vez porque entonces trabajaban en el laboratorio marino de Plymouth, en el Reino Unido (y no es ético realizar experimentos con seres humanos vivos).

El mecanismo del potencial de acción, nombre que recibe el pulso eléctrico que se propaga a lo largo del axón, depende de la apertura de canales de iones, o poros, en la membrana celular de la neurona. Esos canales permiten que los iones de sodio, Na^+ , penetren en el axón y causen un aumento del potencial de la membrana, que al ser grasa ejerce de aislante entre el medio intracelular y el extracelular. Hodgkin y Huxley se dieron cuenta de que una reducción del potencial de la

¹²² Fred Rieke, David Warland, Rob de Ruyter van Steveninck, y William Bialek. *Spikes: Exploring the Neural Code*. MIT Press, 1999. ISBN 0262181746

¹²³ Alan Lloyd Hodgkin y Andrew Fielding Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, 1952. ISSN 0022-3751. DOI: 10.1113/jphysiol.1952.sp004764. URL <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413/>

membrana causaba la activación en cascada de los canales de iones, lo que permite la generación de una señal eléctrica que se propaga de un extremo del axón al otro. Algo así como si encendemos la mecha de un petardo (o de un cartucho de dinamita en una película del Oeste).

En el modelo de Hodgkin-Huxley, el potencial de la membrana V_M viene dado por:

$$C_M \frac{dV_M}{dt} = -g_{Na}(V_M - E_{Na}) - g_K(V_M - E_K) - g_{leak}(V_M - E_{leak})$$

donde g indica la conductancia de los canales de sodio (Na), potasio (K) y de pérdida o escape (*leak*), siendo la conductancia la inversa de la resistencia R en el circuito eléctrico asociado al modelo de Hodgkin y Huxley. Los potenciales E corresponden a los gradientes electroquímicos (voltajes) asociados al flujo de iones a través de la membrana: los debidos a los canales de sodio (E_{Na}) y potasio (E_K), así como el asociado a la corriente de pérdida (E_{leak}). Por último, C_M es la capacidad asociada a la membrana como condensador. El resultado es el circuito RC de la figura, donde la membrana ejerce de condensador y los canales de iones de resistencias.

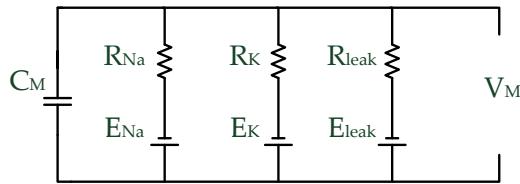


Figura 43: Modelo de Hodgkin-Huxley.

Cuando se concatenan múltiples segmentos del axón modelados de esta forma, aparece el fenómeno de la propagación del potencial de acción o pulso nervioso [*spike*]. Aunque en el circuito eléctrico un pulso se propagaría de la misma forma en ambos sentidos, en una neurona biológica el pulso es principalmente unidireccional: cuando un canal de sodio se cierra, no se puede reabrir inmediatamente.

Aunque un neurocientífico trabaje habitualmente con modelos a este nivel de detalle, en Inteligencia Artificial podemos trabajar con un mayor nivel de abstracción. Si ignoramos los canales de iones, obtenemos un modelo de neurona estándar: el modelo de integración y disparo con pérdidas [*leaky integration-and-fire neuron model*].

$$C_M \frac{dV_j}{dt} = -g_{leak}(V_j - E_{leak}) + g_{int} \sum_{i=1}^n x_i w_{ij}$$

V_j representa el potencial de la ‘membrana’ para la neurona j -ésima, x_i la salida actual de la neurona i -ésima y w_{ij} el peso de la conexión desde la neurona i hasta la neurona j . En la ecuación también aparecen dos conductancias constantes: la constante de pérdida, g_{leak} , y la constante de integración, g_{int} .

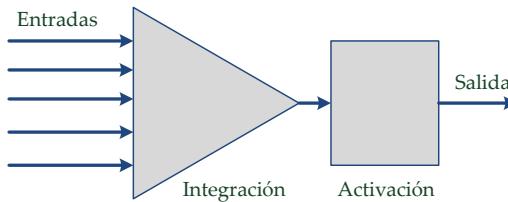
Si asumimos que $E_{leak} = 0$ y anulamos los factores de escala asociados a las conductancias ($g_{leak} = g_{int} = 1$):

$$C_M \frac{dV_j}{dt} = -V_j + \sum_{i=1}^n x_i w_{ij}$$

Si ignoramos la corriente de pérdida ($-V_j$), llegamos a un modelo de neurona más simple: el modelo de integración y disparo [*integrate-and-fire*]. Este modelo, sin embargo, es problemático porque las neuronas pueden acumular carga pero nunca la pierden, por lo que tienden a saturarse. Una vez saturada, la neurona generaría siempre la misma salida. Si, tras eliminar la corriente de pérdida que descargaría el condensador, la sustituimos por una anulación de su carga tras cada intervalo de tiempo, obtenemos un modelo de neurona sin memoria, en el que su voltaje de salida depende única y exclusivamente de la suma ponderada de pesos y entradas (sin que intervenga en ningún momento el valor de la neurona en el instante de tiempo anterior):

$$V_j = \sum_{i=1}^n x_i w_{ij}$$

En Inteligencia Artificial, éste será el modelo que normalmente utilicemos como base para construir redes neuronales artificiales.



Nuestro modelo simplificado de neurona artificial consta de dos etapas. En una primera etapa, se combinan las entradas provenientes de otras neuronas teniendo en cuenta los pesos de las sinapsis. El resultado de esta primera etapa es la entrada neta o excitación de la neurona. En la segunda etapa, la entrada neta se utiliza directamente para determinar el valor de salida de la neurona, que se propagará a otras neuronas.

En ocasiones, nos interesará que el comportamiento de la neurona esté sesgado. El sesgo ejerce, en una neurona artificial, el papel de las corrientes de pérdida en una neurona biológica. Matemáticamente, podemos añadir ese sesgo como un parámetro interno que depende del estado de la neurona, p.ej.

$$\Delta V_j = -V_j + \sum_{i=1}^n x_i w_{ij}$$

En la ecuación anterior, el estado de la neurona en un momento determinado influye al actualizar su estado cuando recibe nuevas entradas.

Figura 44: Modelo simplificado de neurona con integración y activación.

Aunque es una forma relativamente sencilla de dotar a nuestra neurona de cierta capacidad de memoria, la mayoría de los modelos de redes neuronales artificiales utilizan un sesgo externo, o *bias*, b_j :

$$V_j = b_j + \sum_{i=1}^n x_i w_{ij}$$

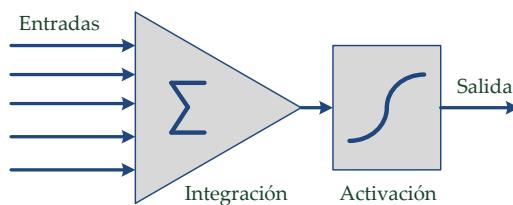
Este sesgo es un parámetro adicional de la neurona, igual que los pesos. De hecho, a menudo se asume que el sesgo no es más que un peso adicional vinculado a una entrada fija con valor 1. Esto es, si establecemos $w_{0j} = b_j$ y $x_0 = 1$, la expresión anterior se puede simplificar:

$$V_j = \sum_{i=0}^n x_i w_{ij}$$

Aunque hasta ahora hemos supuesto que la combinación de entradas y pesos generaba directamente la salida, en el modelo general de neurona artificial, la generación de la salida de la neurona es responsabilidad de una fase de activación independiente de la fase de integración de entradas. Generalmente, la activación de una neurona artificial no reproduce sin más su entrada neta, sino que aplica algún tipo de transformación no lineal a esa entrada neta.

¿Por qué no lineal? Porque cuando construyamos redes con múltiples capas, si las neuronas fuesen siempre lineales, el resultado final del cálculo realizado por la red seguiría siendo una función lineal de las entradas. En otras palabras, cualquier red neuronal compuesta únicamente por elementos lineales siempre se podría sustituir por una red formada por una única capa de neuronas lineales. Las neuronas lineales, obviamente, sólo pueden representar funciones lineales. Como la mayor parte de los problemas reales incorporan algún tipo de no linealidad, nos interesarán que nuestras redes sean capaces de representar esas no linealidades.

Llegamos, por fin, al modelo estándar de neurona artificial utilizado en redes neuronales artificiales. Esas neuronas, en ocasiones, reciben el nombre de elementos de procesamiento [*PEs: Processing Elements*]. Hay quien incluso aboga por utilizar un término como ‘neurodo’ [*neurode*] para dejar clara la diferencia entre una neurona artificial y una neurona biológica. Afortunadamente, esa propuesta no ha llegado a cuajar.



Habitualmente, las redes neuronales artificiales están formadas por conjuntos de neuronas que agruparemos en capas, de forma que todas

Figura 45: Modelo estándar de neurona artificial con función de activación no lineal.

las neuronas de una misma capa comparten ciertas características. Por convención, denotaremos por x_i a cada una de las n entradas recibidas por una capa de neuronas formada por m neuronas. La salida de cada neurona la representaremos por y_j , habitualmente un valor binario o real, que podemos considerar análogo al estado de activación (binario) o a la frecuencia de activación (real) de una neurona biológica.

Los pesos sinápticos w_{ij} los utilizaremos para modelar las conexiones de entrada a las neuronas, siendo el peso w_{ij} el peso asociado a la sinapsis que conecta la entrada i -ésima con la neurona j -ésima. Los pesos, por regla general, tendrán valores reales, positivos para modelar conexiones excitatorias y negativos para modelar conexiones inhibitorias.

Como ya se comentó antes, en ocasiones asumiremos que todas las neuronas reciben una entrada especial permanentemente fija, $x_0 = 1$. Esa entrada nos permitirá modelar el sesgo o *bias* b_j de la neurona j como un peso más: $w_{0j} = b_j$. En determinados modelos de neuronas binarias, el sesgo servirá para determinar el umbral de activación de la neurona, el punto a partir del cual la neurona activa su salida. Si el nivel de excitación de la neurona queda por debajo del umbral, se mantiene inactiva su salida. En cuanto su entrada neta supera el umbral de activación, se activa la salida de la neurona.

En la etapa de integración de entradas, una neurona artificial combina las diferentes entradas x_i con sus pesos para determinar su entrada neta z_j o net_j :

$$z_j = net_j = \sum_i w_{ij}x_i$$

En la etapa de activación, una neurona artificial utiliza el valor asociado a su entrada neta para generar una salida y_j . El modelo más habitual genera una salida de tipo numérico a partir de la entrada neta de la neurona:

$$y_j = f(z_j) = f(net_j) = f\left(\sum_i w_{ij}x_i\right)$$

donde la función f es la función de activación de la neurona, usualmente no lineal.

En otros modelos de redes neuronales, la salida puede depender del valor de activación previo de la neurona. En ese caso, la salida tiene una componente temporal:

$$y_j(t) = F(y_j(t-1), z_j(t)) = F(y_j(t-1), net_j(t))$$

También existen modelos de redes neuronales estocásticas. En ese tipo de modelos, la entrada neta de la neurona se utiliza para representar la probabilidad de activación de una neurona. Si tenemos una neurona estocástica binaria, la misma expresión que utilizábamos antes para la

salida la podemos emplear para establecer una probabilidad de activación p_j :

$$p_j = f(z_j) = f(\text{net}_j) = f\left(\sum_i w_{ij}x_i\right)$$

En este caso, una neurona estocástica binaria generaría un pulso (salida 1) con probabilidad p_j y se mantendría inactiva (salida 0) con probabilidad $1 - p_j$.

El uso continuado de subíndices en las expresiones anteriores puede haberle parecido algo excesivo, tal vez incluso molesto. Ya que estábamos modelando neuronas individuales, ¿por qué la manía de ir siempre arrastrando un subíndice j ? Como ya comentamos, las redes neuronales artificiales se construyen interconectando bloques, como si de piezas de LEGO se tratase. Cada uno de esos bloques, en vez de contener una única neurona, contiene un conjunto de neuronas similares que operan conjuntamente: una capa de neuronas.

Cuando diseñamos una red neuronal artificial compuesta por varias capas de elementos de procesamiento idénticos, el uso de la notación vectorial puede simplificar determinadas expresiones. Por ejemplo, en vez de definir la salida de una única neurona y_j , podemos emplear la notación vectorial para definir simultáneamente las salidas de todas las neuronas de una capa de neuronas:

$$y = f(z) = f(Wx)$$

donde y es el vector de salida de la capa, de tamaño m (el número de neuronas de la capa); x es el vector de entradas, de tamaño n ; y, por último, W es la matriz de pesos, de tamaño $m \times n$.

Los sesgos de las neuronas, b_j , podríamos incorporarlos fácilmente a la expresión anterior:

$$y = f(z) = f(Wx + b)$$

O bien asumir que existe una entrada fija, $x_0 = 1$, y los sesgos van ya representados en la matriz de pesos W , que entonces tendría tamaño $m \times (n + 1)$.

A parte de su economía expresiva, la notación vectorial ofrece otra ventaja muy importante en la práctica. Nos permite paralelizar con facilidad muchos de los algoritmos que se emplean para entrenar y utilizar redes neuronales artificiales. Ya sea mediante el juego de instrucciones SIMD de nuestro microprocesador (MMX, SSE o AVX) o, preferiblemente, usando capacidad de cálculo vectorial de una GPU, la notación vectorial nos indica directamente cómo realizar una implementación paralela de los algoritmos que trabajan con redes neuronales artificiales.

Ya estamos en condiciones de comenzar a analizar los detalles de tipo algorítmico que nos permitirán resolver problemas usando redes neurona-

les artificiales. Empecemos con las funciones de activación utilizadas en distintos modelos de neuronas artificiales:

Funciones de activación

Los modelos de neuronas utilizados en redes neuronales artificiales combinan sus entradas usando pesos que modelan sus conexiones sinápticas y, a continuación, le aplican a la entrada neta de la neurona una función de activación o transferencia. La entrada neta de la neurona recoge el nivel de estímulo que la neurona recibe de sus entradas y es la función de activación la que determina cuál es la salida de la neurona.

A grandes rasgos, las funciones de activación se pueden clasificar en:

- *Funciones de activación discretas*

La salida de la neurona es discreta; esto es, sólo puede tomar un conjunto finito de valores. Normalmente, se utilizan dos valores, por lo que hablamos de neuronas binarias cuando la salida es 0 ó 1, mientras que hablamos de neuronas bipolares cuando su salida puede ser -1 ó +1. Los primeros modelos de redes neuronales artificiales eran discretas, como es el caso de las TLU [*Threshold Logic Unit*] usadas por el algoritmo de aprendizaje del perceptrón.

Las neuronas estocásticas utilizadas en algunos modelos de redes, como las máquinas de Boltzmann, suelen ser binarias, si bien internamente utilizan una función de activación continua que se emplea para determinar la probabilidad con la que se activará la neurona. Incluso las neuronas de las máquinas de Boltzmann continuas son binarias: admiten entradas continuas, pero su salida es binaria (y estocástica).

- *Funciones de activación continuas*

En este caso, la salida de la neurona puede tomar cualquier valor dentro de un intervalo. Generalmente, el rango de ese intervalo está limitado, o bien al intervalo [0, 1] o bien al intervalo [-1, 1].

Dada una función de activación $f_{[0,1]}$ con un rango definido, el intervalo [0, 1], es fácil crear una función de activación con el rango deseado $[a, b]$. Basta con aplicar una transformación lineal del tipo:

$$f_{[a,b]}(z) = a + (b - a)f_{[0,1]}(z)$$

Resulta trivial crear versiones bipolares de cualquier función de activación que sólo tome valores positivos:

$$f_{[-1,1]}(z) = 2f_{[0,1]}(z) - 1$$

Dentro de las funciones de activación continuas, se pueden utilizar tanto funciones de activación lineales (p.ej. la función identidad) como

funciones no lineales. Éstas últimas son las más utilizadas habitualmente, ya que son las que dotan a una red neuronal multicapa de su capacidad de aproximador universal.

Función de activación lineal

La función de activación más sencilla que podemos imaginarnos para una neurona artificial es la función identidad:

$$y = f_{lin}(z) = z$$

Sobre esta función de activación podemos aplicar cualquier transformación lineal de las mencionadas arriba y seguiríamos teniendo una neurona lineal.

La característica principal de las neuronas lineales es que, si conectamos varias capas de neuronas lineales en serie, el resultado final será siempre equivalente a una única capa de neuronas lineales. Imaginemos que tenemos una capa de neuronas con una matriz de pesos W_1 . Su salida la utilizamos como entrada de una segunda capa de neuronas, con pesos W_2 . El resultado de la composición será:

$$y = f_2(f_1(x)) = W_2(W_1x) = (W_2W_1)x$$

Es decir, podemos sustituir ambas capas por una única capa con una matriz de pesos $W = W_2W_1$.

Desde un punto de vista más formal, las capas de neuronas lineales actúan como simples clasificadores lineales, por lo que no resultan adecuadas para resolver problemas complejos por sí mismas. Ahora bien, aunque pueda parecer que las limitaciones de las neuronas lineales hacen que no se deban utilizar en la práctica, esto no es del todo cierto. En primer lugar, existen problemas sencillos para los que redes neuronales artificiales simples pueden ser más que suficientes (aunque generalmente nadie las llamará redes neuronales). En segundo lugar, hay ocasiones en las que las neuronas lineales pueden utilizarse como componentes de una red más grande, que incorporará otros componentes no lineales. Es el caso de las redes convolutivas que se utilizan a menudo para trabajar con imágenes. En tercer lugar, el uso de múltiples capas consecutivas de unidades lineales puede tener sentido como una forma de factorizar la matriz de pesos, reduciendo el número de parámetros que hemos de ajustar. Por ejemplo, si tenemos una primera capa con a entradas y b salidas seguida de una segunda etapa de b entradas y c salidas, tendremos que ajustar $ab + bc = (a + c)b$ parámetros frente a los ac parámetros de la capa combinada. Si b es pequeño con respecto a a y c , el ahorro puede ser considerable.

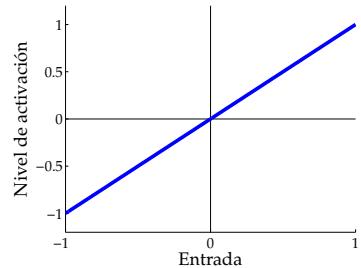


Figura 46: Función de activación lineal.

Función escalón

La primera función no lineal que se utilizó para modelar neuronas artificiales fue la función escalón [*step function*], también conocida como función umbral [*threshold function*], limitador estricto [*hard limiter*] o función de Heaviside, en honor al matemático, físico e ingeniero inglés que la utilizó para analizar comunicaciones telegráficas. Ya en redes neuronales artificiales, es la función usada en las unidades TLU [*Threshold Logic Units*] del modelo neuronal de McCulloch y Pitts:

$$y = f_{tlu}(z) = u(z) = \mathbf{1}_{z \geq 0} = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{si } z < 0 \end{cases}$$

Obviamente, se trata de una función de activación binaria de la que podemos derivar una versión bipolar: la función signo [*signum function*], también conocida como limitador estricto simétrico [*symmetric hard limiter*].

$$y = f_{sgn}(z) = sgn(z) = \begin{cases} 1 & \text{si } z \geq 0 \\ -1 & \text{si } z < 0 \end{cases}$$

Esta función de activación es simétrica con respecto al origen. Se utilizó, en algunos modelos primitivos de redes neuronales artificiales, porque su implementación electrónica resultaba más sencilla que en su versión binaria.

Función lineal con saturación

Si, en lugar de una función de activación discreta, ya sea binaria o bipolar, queremos disponer de una función de activación continua, podemos diseñar fácilmente una función lineal de activación a trozos que limite el rango de salida de la neurona:

- En su versión binaria:

$$y = f_{sl}(z) = \begin{cases} 1 & \text{si } z > \frac{1}{2} \\ z + \frac{1}{2} & \text{si } -\frac{1}{2} \leq z \leq \frac{1}{2} \\ 0 & \text{si } z < -\frac{1}{2} \end{cases}$$

- En su versión bipolar o simétrica:

$$y = f_{ssl}(z) = \begin{cases} 1 & \text{si } z > 1 \\ z & \text{si } -1 \leq z \leq 1 \\ -1 & \text{si } z < -1 \end{cases}$$

Una forma alternativa de definir esta función es la siguiente:

$$y = f_{ssl}(z) = \max(-1, \min(1, z))$$

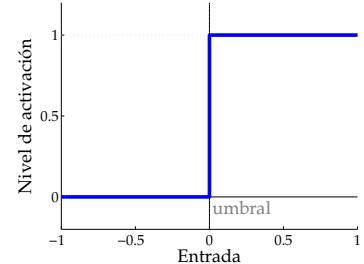


Figura 47: Función de activación TLU.

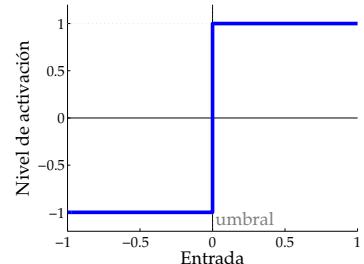


Figura 48: Función de activación TLU simétrica o bipolar.

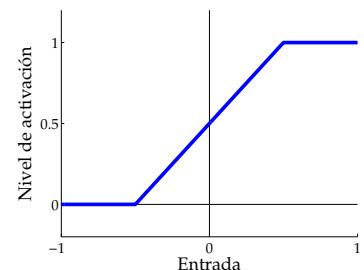


Figura 49: Función de activación lineal con saturación.

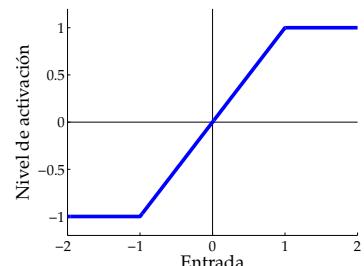


Figura 50: Función de activación lineal bipolar con saturación.

Por su similitud con la tangente hiperbólica, Ronan Collobert la denominó tangente hiperbólica estricta [*hard tanh*]. Su derivada, aunque no está formalmente definida para los puntos $z = 1$ y $z = -1$, puede calcularse fácilmente en la práctica:

$$\frac{d}{dz} f_{ssl}(z) = \begin{cases} 1 & \text{si } |z| \leq 1 \\ 0 & \text{en otro caso} \end{cases}$$

Como veremos más adelante, la derivada de la función de activación desempeña un papel fundamental a la hora de poder ajustar los parámetros de una red neuronal durante su entrenamiento, por lo que a menudo recurriremos a funciones de activación cuya derivada esté definida para cualquier valor y cumpla algunas propiedades “deseables”.

Función de activación sigmoidal

Usualmente, nos interesará que la función de activación de una neurona sea, además de no lineal, estrictamente creciente, continua y derivable. Las funciones sigmoidales satisfacen todos esos requisitos, lo que las hace especialmente útiles en redes neuronales que se entrena usando *backpropagation*. *Backpropagation* es un algoritmo de propagación de errores que, dados los errores observados en la capa de salida, propaga esos errores hacia atrás en la red para ir ajustando sus parámetros internos. La forma en que se ajustan esos parámetros depende de la derivada de la función de activación de las neuronas, de ahí nuestro interés en que la función sea derivable.

Existen distintas funciones sigmoidales. Todas tienen en común su característica forma de ‘S’. Algunas de ellas tienen propiedades matemáticas que las hacen especialmente interesantes para su uso en redes neuronales artificiales. Cuando existe una relación sencilla entre el valor de la función en un punto y el valor de su derivada en ese punto, podemos aprovechar esa relación para reducir el coste computacional del entrenamiento de la red neuronal: una vez calculado el valor de activación de la neurona, una sencilla expresión aritmética nos permite obtener su derivada de forma eficiente.

Veamos algunos ejemplos de funciones sigmoidales: la función logística, la tangente hiperbólica y la función gudermanniana.

- *Función logística*

$$y = f_{logistic}(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

La función logística es simétrica, en el sentido de que

$$\sigma(-z) = 1 - \sigma(z)$$

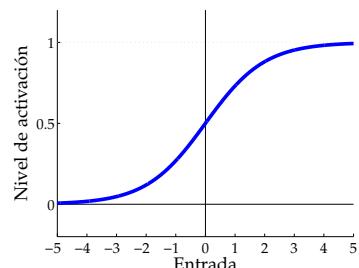


Figura 51: Función de activación sigmoidal: Función logística.

La función logística es una sigmoide “binaria”, con rango $[0, 1]$, cuya derivada calculamos a continuación:

$$\begin{aligned}\frac{d\sigma(z)}{dz} &= \frac{d}{dz} \left[\frac{1}{1+e^{-z}} \right] \\ &= \frac{0(1+e^z) - 1(-e^z)}{(1+e^{-z})^2} \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1}{1+e^{-z}} \frac{e^{-z}}{1+e^{-z}}\end{aligned}$$

Ahora bien,

$$\frac{e^{-z}}{1+e^{-z}} = \frac{(1+e^{-z})-1}{1+e^{-z}} = \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} = 1 - \sigma(z)$$

Por tanto:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

La relación entre el valor de la función $\sigma(z)$ y su derivada nos permite calcular el valor de esta última con sólo dos operaciones aritméticas, una resta y una multiplicación, una vez que conocemos el valor de $\sigma(z)$. Sin necesidad de realizar llamadas a funciones trascendentales, cuya evaluación es usualmente más costosa desde el punto de vista computacional.

La función logística se obtiene como solución de la siguiente ecuación diferencial ordinaria y no lineal de primer orden:

$$\frac{d}{dx}f(x) = f(x)(1 - f(x))$$

al establecer como condición inicial $f(0) = 1/2$.

Esta ecuación aparece en muchas ramas de la ciencia, ya que nos permite, entre otras muchas cosas, modelar el crecimiento de poblaciones en ecología, hacer un seguimiento de la concentración de reactivos y productos en reacciones autocatalíticas, o calcular la distribución estadística de los estados de energía de los fermiones en equilibrio térmico.

Desde un punto de vista más algorítmico, puede que nos interese controlar la pendiente de la función logística para que el máximo de su derivada no se quede en $1/4$ (cuando $z = 0$). Para ello, podemos introducir un parámetro s que controle la pendiente de la sigmoide:

$$y = f_{sigmoid}(z) = \frac{1}{1 + e^{-sz}}$$

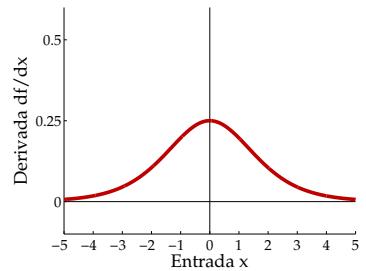


Figura 52: Derivada de la función de activación sigmoidal: Derivada de la función logística.

Una función trascendente, a veces denominada trascendental por traducción directa del inglés, es una función que trasciende al álgebra en el sentido de que no puede ser expresada en términos de una secuencia finita de operaciones algebraicas de suma, resta, multiplicación, división y extracción de raíces. El logaritmo, la exponentiación y las funciones trigonométricas son funciones trascendentes, mientras que la raíz cuadrada no lo es (la raíz cuadrada es una función algebraica que satisface la ecuación polinómica $r^2 = x$).

cuya derivada es

$$\frac{d}{dz} f_{sigmoid}(z) = \frac{e^{-sz}}{(1+e^{-sz})^2} = s f_{sigmoid}(z)(1 - f_{sigmoid}(z))$$

Realizando una sencilla transformación lineal, también podemos obtener una sigmoide “bipolar”, con rango $[-1, +1]$:

$$y = f_{bipolar}(z) = 2f_{sigmoid}(z) - 1 = \frac{2}{1+e^{-sz}} - 1 = \frac{1-e^{-sz}}{1+e^{-sz}}$$

La derivada de esta función sigmoidal bipolar es:

$$\begin{aligned} \frac{d}{dz} f_{bipolar}(z) &= 2 \frac{d}{dz} f_{sigmoid}(z) \\ &= 2s f_{sigmoid}(z)(1 - f_{sigmoid}(z)) \end{aligned}$$

o, de forma alternativa, en términos de la propia función bipolar:

$$\frac{d}{dz} f_{bipolar}(z) = \frac{s}{2} (1 + f_{bipolar}(z)) (1 - f_{bipolar}(z))$$

Como caso particular de esta función sigmoidal bipolar, cuando $s = 2$, nos encontramos con otra de las funciones de activación más utilizadas: la tangente hiperbólica.

■ Tangente hiperbólica

La tangente hiperbólica se utiliza en trigonometría esférica y se define de la siguiente forma:

$$y = f_{tanh}(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{1 - e^{-2z}}{1 + e^{-2z}}$$

Tal como acabamos de ver, no es más que una versión bipolar de la función sigmoidal derivada de la función logística con el parámetro $s = 2$. Esto es, la tangente hiperbólica está estrechamente relacionada con la función logística:

$$\tanh(z) = 2 \sigma(2z) - 1$$

La derivada de la tangente hiperbólica en un punto también se puede expresar directamente en términos del valor de la función en ese punto:

$$\frac{d \tanh(z)}{dz} = (1 + \tanh(z))(1 - \tanh(z)) = (1 - \tanh^2(z))$$

Igual que sucedía con la función logística, la tangente hiperbólica también aparece en otros dominios científicos como solución a una ecuación diferencial no lineal

$$\frac{d}{dx} f(x) = 1 - f^2(x)$$

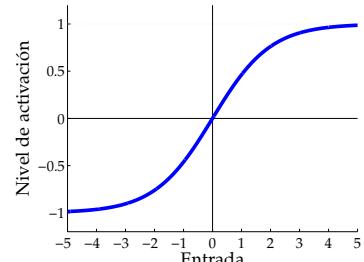


Figura 53: Función de activación sigmoidal bipolar: Versión bipolar de la función logística.

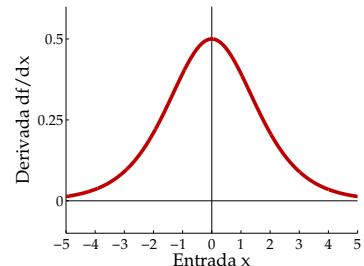


Figura 54: Derivada de la función de activación sigmoidal bipolar.

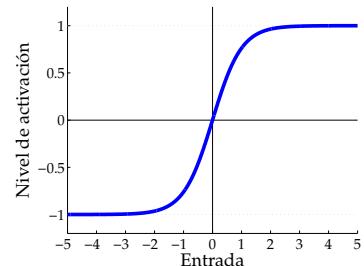


Figura 55: Función de activación sigmoidal bipolar: Tangente hiperbólica.

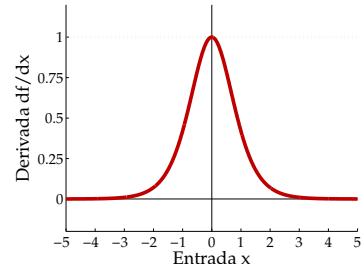


Figura 56: Derivada de la tangente hiperbólica. Obsérvese el efecto que tiene el uso del parámetro $s = 2$.

bajo las condiciones iniciales $f(0) = 0$ y $f'(\infty) = 0$.

La tangente hiperbólica, según algunos autores, funciona mejor que la función logística al parecerse más a la función identidad para valores de z cercanos a cero. Recuerde que $\tanh(0) = 0$ mientras que $\sigma(0) = 1/2$. Mientras los niveles de activación de las neuronas de la red se mantengan pequeños, lo que se puede conseguir con la ayuda de técnicas de regularización, el comportamiento de las unidades con una función de activación tanh es similar a las de las unidades lineales, lo que puede facilitar el entrenamiento de la red neuronal.

■ Función gudermanniana

Para concluir esta sección sobre funciones sigmoidales, mencionaremos una tercera función de activación: la función gudermanniana.

La función gudermanniana recibe su nombre en honor de Christoph Gudermann, matemático alemán del siglo XIX, y profesor de Karl Weierstrass. Como suele suceder, Gudermann no fue el primero que definió esta función. De hecho, la función fue denominada anteriormente ‘ángulo trascendental’ por el suizo Johann Heinrich Lambert, que fue precisamente quien incorporó las funciones hiperbólicas a la trigonometría en el siglo XVIII, además de crear el higrómetro moderno, un instrumento que se utiliza para medir la humedad atmosférica.

¿Qué cómo se define la función gudermanniana? Aquí está su definición matemática:

$$y = f_{gd}(z) = \text{gd } z = \int_0^z \frac{1}{\cosh t} dt$$

La función gudermanniana, tal como se ve en la figura, toma valores de $-\pi/2$ a $\pi/2$. Tras ver su definición, estará pensando cómo implementarla algorítmicamente de forma eficiente. En un momento salimos de dudas.

La función gudermanniana relaciona las funciones trigonométricas, también conocidas como funciones circulares, con las funciones hiperbólicas sin recurrir a los números complejos. Las siguientes expresiones tal vez le ayuden:

$$\text{gd } x = \arcsin(\tanh x) = 2 \arctan \left[\tanh \left(\frac{x}{2} \right) \right] = 2 \arctan(e^x) - \frac{\pi}{2}$$

De acuerdo, era una broma... Su cálculo involucra varias funciones trascendentales, por lo que no tiene demasiado sentido emplearla en redes neuronales artificiales cuando ya tenemos otras funciones con las mismas características y propiedades que las hacen especialmente adecuadas para conseguir implementaciones altamente eficientes.

Aunque no se emplee en redes neuronales por motivos obvios, la función gudermanniana sí que aparece en geometría hiperbólica (en la definición del ángulo de paralelismo), al estudiar el efecto Casimir en mecánica cuántica y en la resolución de determinados problemas

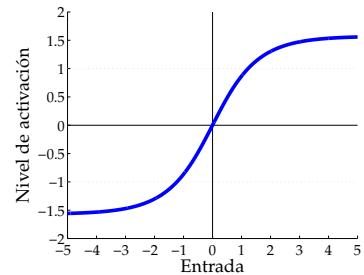


Figura 57: Otra función de activación sigmoidal bipolar con rango de $-\pi/2$ a $\pi/2$: La función gudermanniana.

mecánicos. En particular, aparece en soluciones analíticas no periódicas del péndulo invertido, un problema dinámico clásico que se emplea para evaluar sistemas de control, algunos de los cuales sí que se diseñan utilizando redes neuronales.¹²⁴

Función de activación lineal rectificada

Es posible diseñar una función de activación no lineal que no requiera la utilización de funciones trascendentales. De hecho, se puede diseñar sin que ni siquiera tengamos que realizar operaciones de tipo aritmético, por lo que resulta extremadamente eficiente. Es precisamente lo que hace las unidades lineales rectificadas o ReLU [*REctified Linear Units*]. Estas unidades utilizan una función de activación lineal rectificada, que se define de la siguiente forma:

$$y = f_{relu}(z) = \begin{cases} z & \text{si } z \leq 0 \\ 0 & \text{si } z < 0 \end{cases}$$

Igual que sucede con otras funciones de activación, existen formas alternativas de definir la función de activación lineal rectificada:

$$y = f_{relu}(z) = z^+ = \max(0, z)$$

La derivada de una función lineal rectificada es trivial y coincide con la función escalón utilizada por las neuronas de McCulloch y Pitts:

$$\frac{d}{dx} f_{relu}(z) = u(z) = \mathbf{1}_{z \geq 0} = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{si } z < 0 \end{cases}$$

Existe una versión suavizada de la función de activación lineal rectificada: la función *softplus*.¹²⁵ Su nombre proviene, precisamente, del hecho de que esta función, continua y derivable, suaviza el codo asociado a la función z^+ .

$$y = f_{softplus}(z) = \zeta(z) = \log(1 + e^z)$$

Esta función, que se denota con la letra griega zeta (ζ), tiene varias propiedades matemáticas que la relacionan con la función logística:

$$\log \sigma(x) = -\zeta(-x)$$

$$\zeta(x) = \int_{-\infty}^x \sigma(y) dy$$

De hecho, su derivada es la propia función logística:

$$\frac{d}{dx} f_{softplus}(z) = \frac{d}{dx} \zeta(z) = \sigma(z)$$

¹²⁴ W. Thomas Miller III, Richard S. Sutton, y Paul J. Werbos, editores. *Neural Networks for Control. Neural Network Modeling and Connectionism*. MIT Press, 1990. ISBN 0262132613

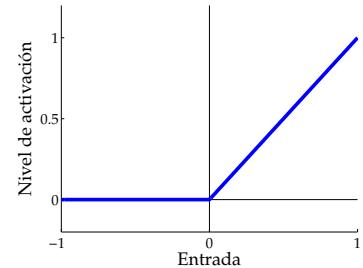


Figura 58: Función de activación lineal rectificada, utilizada por las unidades ReLU, muy habituales en *deep learning*.

¹²⁵ Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, y René Garcia. Incorporating second-order functional knowledge for better option pricing. En *Proceedings of the 13th International Conference on Neural Information Processing Systems, NIPS'00*, pages 451–457, Cambridge, MA, USA, 2000. MIT Press. URL <https://goo.gl/doB3GN>

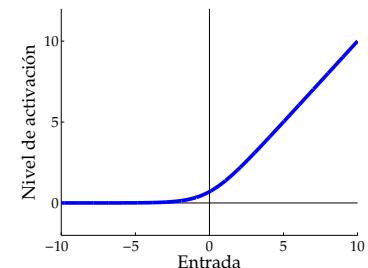


Figura 59: Función de activación *softplus*, una versión suavizada de un rectificador.

Otra propiedad curiosa de la función *softplus* es la siguiente:

$$\zeta(x) + \zeta(-x) = x$$

No es más que una versión suavizada de

$$x^+ - x^- = x$$

Una forma compacta de representar la siguiente tautología:

$$\max(0, x) - \max(0, -x) = x$$

Pese a sus peculiares propiedades matemáticas, la función *softplus* no es más que una ligera variación de la función de activación lineal rectificada. En la práctica, se ha comprobado que las unidades ReLU funcionan mejor.¹²⁶ Una muestra más de que, en ocasiones, menos es más. Aunque carezcan de las propiedades matemáticas de la función *softplus*, las sencillas unidades lineales rectificadas se comportan mejor. Por este motivo, no se suele recomendar el uso de la función *softplus*, salvo que lo que pretendamos sea demostrar analíticamente las propiedades formales de un modelo concreto, para lo cual viene bien disponer de una función continua y derivable.

Existen otras variantes de las unidades lineales rectificadas, tres de las cuales utilizan una pendiente no nula α para valores negativos de z :

$$y = f_{rectification}(z) = \max(0, z) + \alpha \min(0, z)$$

- La rectificación de valor absoluto¹²⁷ utiliza $\alpha = -1$ y se emplea en situaciones en las que no importa la polaridad de la señal de entrada, como en el reconocimiento de objetos en imágenes.

$$y = f_{absolute\ rectification}(z) = \max(0, z) - \min(0, z) = |z|$$

- Las unidades ReLU con pérdidas [*leaky ReLU*] utilizan un pequeño valor de α , p.ej. 0.01, sólo para evitar que la derivada sea siempre cero cuando z es negativo.¹²⁸

$$y = f_{leaky\ relu}(z) = \max(0, z) + 0.01 \min(0, z)$$

- Las unidades ReLU paramétricas, o PReLU [*Parametric ReLU*], interpretan la pendiente α como un parámetro más de la neurona, que hay que aprender (y, obviamente, puede ser diferente para cada neurona de la red).¹²⁹

$$y_j = f_{prelu}(z, \alpha_j) = \max(0, z) + \alpha_j \min(0, z)$$

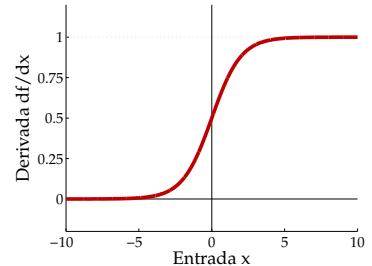


Figura 60: Derivada de la función de activación *softplus*: la función logística.

¹²⁶ Xavier Glorot, Antoine Bordes, y Yoshua Bengio. Deep sparse rectifier neural networks. En Geoffrey Gordon, David Dunson, y Miroslav Dudík, editores, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *JMLR W&CP, Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL <http://proceedings.mlr.press/v15/glorot11a>

¹²⁷ Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, y Yann LeCun. What is the best multi-stage architecture for object recognition? En *IEEE 12th International Conference on Computer Vision, ICCV 2009, Kyoto, Japan, September 27 - October 4, 2009*, pages 2146–2153, 2009. ISBN 978-1-4244-4420-5. DOI: 10.1109/ICCV.2009.5459469

¹²⁸ Andrew L. Maas, Awni Y. Hannun, y Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. En *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013. URL <https://goo.gl/5x3Cj6>

¹²⁹ Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv e-prints*, 2015a. URL <http://arxiv.org/abs/1502.01852>

Aún se pueden diseñar más generalizaciones de las unidades lineales rectificadas para aplicaciones específicas, como es el caso de las unidades *maxout*. Una unidad de este tipo agrupa sus entradas en subgrupos y genera una salida para cada subgrupo.¹³⁰

$$y_k = g(z)_k = \max_{j \in G^{(k)}} z_j$$

Una unidad *maxout* es capaz de aprender una función lineal a trozos, por lo que puede verse como una forma de que la red aprenda su propia función de activación, no sólo los pesos sinápticos asociados a las conexiones entre unas neuronas y otras. Esto dota a la red de cierta redundancia, lo que permite tratar de atajar un fenómeno conocido como “olvido catastrófico”, en el que la red neuronal puede llegar a olvidar cómo realizar tareas para las que ya había sido entrenada en el pasado.¹³¹

En general, las unidades lineales rectificadas tienen una ventaja con respecto a las unidades sigmoidales cuando forman parte de una red que se entrena con algoritmos basados en *backpropagation*. Las unidades sigmoidales se saturan a partir de cierto valor, lo que hace que la derivada de su función de activación sea prácticamente nula. En el entrenamiento basado en el gradiente descendente con *backpropagation*, se utiliza el valor de esa derivada como parte del cálculo del gradiente del error. Ese gradiente es el que determina cómo modificar los pesos de la red. Cuando la derivada de la función de activación es virtualmente nula, la magnitud de las actualizaciones de los pesos será muy pequeña y el entrenamiento de la red avanzará muy lentamente. La saturación prematura de las neuronas sigmoidales puede ocasionar serios problemas en el entrenamiento de una red neuronal. No obstante, si se utiliza una función de coste o pérdida que compense esa saturación, sí se pueden seguir utilizando neuronas sigmoidales.

También hay ocasiones en las que el uso de unidades ReLU se descarta, al no estar su rango acotado, como sucede en redes recurrentes (para evitar una realimentación positiva, que haría que la red quedase fuera de control) o en modelos estocásticos (en los que las funciones de activación se emplean para determinar probabilidades, obviamente acotadas).

Funciones de base radial

No todos los modelos de redes neuronales se basan en el modelo estándar de integración seguida de activación. En ocasiones, puede resultar de interés que una neurona compare su vector de entradas con su vector de pesos y proporcione una salida en función de la similitud entre ambos. Es el caso de las funciones de base radial [*RBFs: Radial Basic Functions*].

Una forma de medir la similitud entre el vector de entradas y el vector de pesos es calcular su distancia euclídea, también conocida como norma

¹³⁰ Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, y Yoshua Bengio. Maxout networks. En *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16–21 June 2013*, pages 1319–1327, 2013b. URL <http://jmlr.org/proceedings/papers/v28/goodfellow13.html>

¹³¹ Ian J. Goodfellow, Mehdi Mirza, Xia Da, Aaron C. Courville, y Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv e-prints*, 2013a. URL <http://arxiv.org/abs/1312.6211>

L^2 de la diferencia entre ambos vectores. Es lo que hacen, por ejemplo, las funciones de base radial de tipo gaussiano:

$$y = f_{rbf}(w, x) = \phi(\|w - x\|) = e^{-\frac{1}{\sigma^2} \|w-x\|^2}$$

La salida de esa función es cero para la mayor parte de los valores de x , por lo que una red con elementos de este tipo puede ser difícil de entrenar utilizando técnicas basadas en el gradiente descendente.

Arquitecturas de las redes neuronales artificiales

Una vez que hemos analizado con detalle algunos de los modelos más utilizados de neuronas artificiales, es el momento de comenzar a ver cómo podemos combinar colecciones de neuronas para resolver problemas de interés.

Redes feed-forward

La mayor parte de las aplicaciones comerciales en las que se han utilizado redes neuronales emplean redes con múltiples capas de neuronas. En su topología más habitual, las neuronas de cada capa suelen ser independientes entre sí y operan en paralelo, lo que facilita su implementación eficiente si disponemos de una GPU. Las distintas capas se conectan entre sí de tal forma que la salida de la capa i se utiliza como entrada en la capa $i + 1$. Estas redes reciben el nombre de redes neuronales *feed-forward*, al carecer de mecanismo alguno de realimentación [*feedback*], tomando prestada la terminología habitual en los sistemas de control. Las redes en las que sí se incluye algún tipo de realimentación de las salidas a las entradas se denominan redes recurrentes.

Las capas de una red multicapa se dividen en dos categorías: capas visibles y capas ocultas. Las capas visibles son aquellas capas (parcialmente) observables desde el exterior de la red. En particular, la primera y la última: las capas de entrada y de salida de la red neuronal. Todas las capas intermedias diferentes a las capas de entrada y de salida reciben el nombre de capas ocultas, por no ser observables directamente desde el exterior.

En una red neuronal de tipo *feed-forward*, nos podemos encontrar con varias situaciones dependiendo del número de capas ocultas que se utilicen:

- *Redes simples, con una única capa*

Es el caso más simple de red neuronal. Las neuronas de la capa de entrada, que se limitan a recibir las señales de entrada provenientes del exterior, redistribuyen esas entradas a las neuronas de la capa de

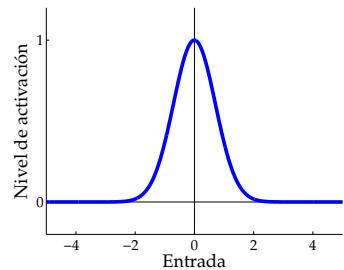


Figura 61: Función de activación RBF de tipo gaussiano.

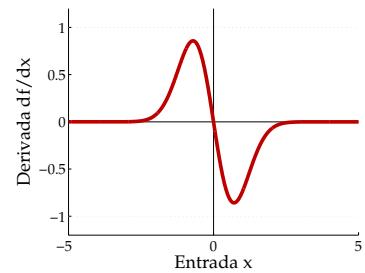


Figura 62: Derivada de una función de activación RBF de tipo gaussiano.

salida, que es la única capa de la red que realmente hace algo. De ahí que, aunque veamos dos capas, una de entrada y una de salida, este tipo de redes se considere unicapa.

Los perceptrones son, tal vez, el modelo más conocido de red neuronal con una sola capa. Fueron las primeras redes neuronales para las que se diseñó un algoritmo de aprendizaje, en los años 50 del siglo XX.

- *Redes multicapa, con una capa oculta*

En una red neuronal simple, tanto la capa de entrada como la capa de salida son visibles desde el exterior de la red. Si añadimos nuevas capas intermedias, estas capas ya no serán visibles desde el exterior, lo que nos obligará a utilizar algoritmos como *backpropagation* para ajustar sus parámetros internos.

La presencia de una única capa oculta ya dota a la red multicapa de su capacidad de aproximador universal (informalmente, su capacidad de aprender cualquier cosa que se pueda aprender). Este tipo de redes fue muy popular desde los años 80 hasta finales del siglo XX.

- *Redes profundas [deep networks], con varias capas ocultas*

Para aprovechar el potencial de las técnicas de *deep learning* a la hora de diseñar soluciones modulares para problemas complejos, las redes neuronales actuales suelen incluir múltiples capas ocultas.

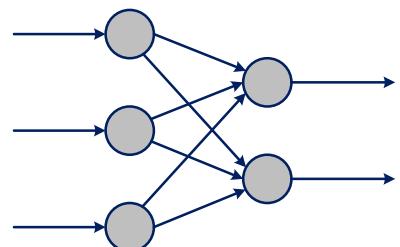


Figura 63: Red neuronal simple, sin capas ocultas. La red sólo tiene una capa de neuronas de entrada y una capa de neuronas de salida.

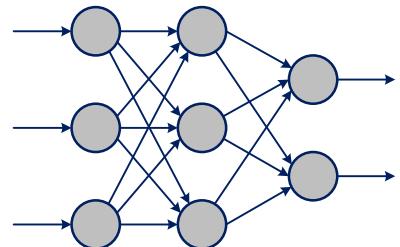
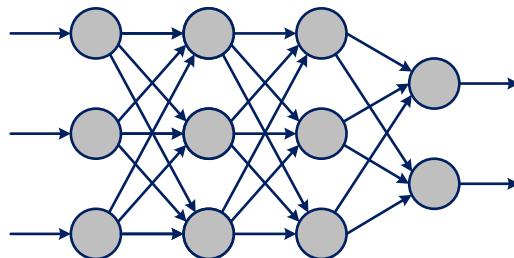


Figura 64: Red neuronal multicapa, con una única capa oculta.



En todas las redes, las conexiones pueden ser excitatorias o inhibitorias, dependiendo del signo del peso asociado a ellas. Las funciones de activación bipolares, como la tangente hiperbólica, pueden resultar útiles en situaciones en las que un cambio de signo en la salida de la neurona influya en el comportamiento de las neuronas de las siguientes capas. Si la salida bipolar de una neurona se conecta a la entrada de otra neurona con un peso sináptico positivo, la conexión será excitatoria cuando la salida sea positiva e inhibitoria cuando sea negativa (o al revés si el peso sináptico es negativo). Otras funciones de activación, como la función logística, carecen de esa propiedad. Ante valores bajos, la neurona simplemente no se activará y no ejercerá ningún tipo de influencia sobre las neuronas de las capas siguientes (ni excitatoria ni inhibitoria). Un

Figura 65: Red neuronal profunda, con múltiples capas ocultas (en este caso, sólo dos).

punto más a favor del uso de la tangente hiperbólica frente a la función logística.

Redes competitivas

Otra topología de red que aparece en ocasiones incluye conexiones inhibitorias entre las neuronas de una misma capa. Es el caso de las redes competitivas.

En una capa competitiva, las conexiones dentro de la misma capa son siempre inhibitorias. En la figura, se muestran con una línea discontinua y terminadas en un círculo. Esas conexiones inhibitorias hacen que, cuando una neurona se active, ejerza una influencia inhibitoria sobre todas las demás. En casos extremos de competitividad, de tipo *winner-takes-all*, sólo una de las neuronas de la capa se activará, permaneciendo las demás inactivas.

Las redes de tipo competitivo son útiles para modelar situaciones en las que una capa deba responder en función del contexto, como sucede en algunos modelos de memorias asociativas. Cada neurona de una capa competitiva se especializará en detectar un patrón de activación específico en la señal de entrada, proveniente de la salida de la capa anterior de la red (la capa aferente desde el punto de vista de la capa competitiva).

Existe infinidad de modelos de redes neuronales artificiales que utilizan alguna forma de señal inhibitoria a la hora de procesar una señal de entrada.

- En algunos modelos, los pesos sinápticos se mantienen fijos, no se entrena. Estos modelos utilizan la competición entre neuronas de la capa competitiva para mejorar el contraste en los niveles de activación de las neuronas. De ahí que se denominen redes de mejora del contraste [*contrast-enhancing networks*].¹³² Algunos ejemplos de este tipo de redes son las redes MAXNET, las redes del sombrero mexicano [*mexican hat networks*] o las redes de Hamming.
- Otros modelos de redes competitivas utilizan algoritmos de aprendizaje supervisado para ajustar sus parámetros, como sucede en las técnicas LVQ [*Learning Vector Quantization*] de Teuvo Kohonen¹³³ o las redes de contra-propagación CPN [*counterpropagation networks*] de Robert Hecht-Nielsen.¹³⁴
- Por último, también existen algunos modelos de redes competitivas que se entrena de forma no supervisada. Los más conocidos son los mapas auto-organizativos SOM [*Self-Organizing Maps*] de Teuvo Kohonen¹³⁵ y las redes basadas en la teoría de la resonancia adaptativa ART [*Adaptive Resonance Theory*] de Stephen Grossberg y Gail Carpenter.¹³⁶

Muchos de estos diseños de redes neuronales artificiales, algunos de los

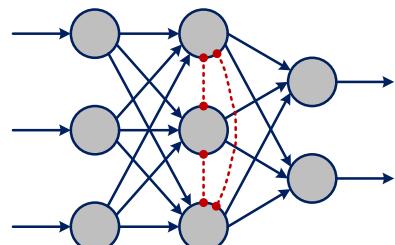


Figura 66: Red neuronal competitiva, con conexiones inhibitorias entre neuronas de la misma capa.

¹³² Laurene V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice Hall, 1994. ISBN 0133341860

¹³³ Teuvo Kohonen. Learning Vector Quantization. En Michael A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 537–540. MIT Press, 1998. ISBN 0262511029

¹³⁴ Robert Hecht-Nielsen. Counter-propagation networks. *Applied Optics*, 26(23):4979–4984, 1987. DOI: 10.1364/AO.26.004979

¹³⁵ Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982. ISSN 1432-0770. DOI: 10.1007/BF00337288

¹³⁶ Gail A. Carpenter y Stephen Grossberg. The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network. *Computer*, 21(3):77–88, 1988. ISSN 0018-9162. DOI: 10.1109/2.33

cuales pueden ser algo bizantinos, están inspirados en el funcionamiento observado de las redes neuronales de tipo biológico. En ellas, se suele luchar contra la tendencia de saturar su capacidad sin añadir información útil añadiendo algún tipo de inhibición que, dada una señal, le reste la media y transmita sólo la diferencia con respecto a la media. Lo vimos al comentar el papel desempeñado por las células horizontales en la retina de un mamífero.

Algunos de los modelos propuestos, como los mapas auto-organizativos de Kohonen, son capaces de emular la configuración espacial observada en muchos circuitos neuronales biológicos.

Es el caso de los mapas tonotópicos asociados al procesamiento de sonidos de diferentes frecuencias, desde la membrana basilar de la cóclea hasta el córtex auditivo primario situado en el lóbulo temporal del cerebro, pasando por el complejo olivar superior del tallo encefálico en el que se recibe la señal del nervio vestibulococlear.

De la misma forma, esta organización espacial se observa en el procesamiento de la información visual, desde los campos receptivos locales de las neuronas de la retina hasta la respuesta espacial de las diferentes regiones del córtex visual situado en el lóbulo occipital del cerebro, pasando por la organización de los clusters neuronales del núcleo geniculado lateral del tálamo, que ejerce de relé entre la retina y el córtex visual preservando la organización bidimensional de la imagen captada en la retina. De hecho, la franja dorsal del córtex visual se especializa en identificar la localización de los objetos y su movimiento (la ventral se ocupa de identificarlos, ignorando la información relativa a su localización exacta). La organización espacial de neuronas que procesan información visual respetando la topografía bidimensional de la imagen también se imita con éxito en las redes neuronales convolutivas, empleadas en visión artificial para reconocer objetos en imágenes y localizarlos.

En realidad, la configuración espacial de muchos circuitos neuronales es consecuencia de un diseño en el que se intenta minimizar su cableado; esto es, la longitud de los axones que conectan unas neuronas con otras. Las neuronas que se activan a la vez tienden a situarse, pues, cerca unas de otras, lo que consigue un diseño más eficiente desde el punto de vista energético. De hecho, esa organización topográfica aparece más allá de las neuronas sensoriales y del córtex visual, auditivo y somatosensorial.

Redes recurrentes

Por su topología, las redes de tipo *feed-forward* carecen por completo de memoria. Cuando se añaden conexiones inhibitorias entre neuronas de una misma capa, se consiguen redes cuyo funcionamiento depende del contexto. Si las conexiones entre neuronas de la misma capa se generalizan (o se admiten conexiones de salida de una capa como entradas de una capa

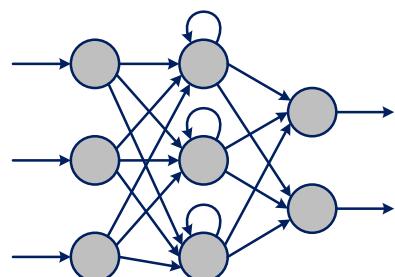
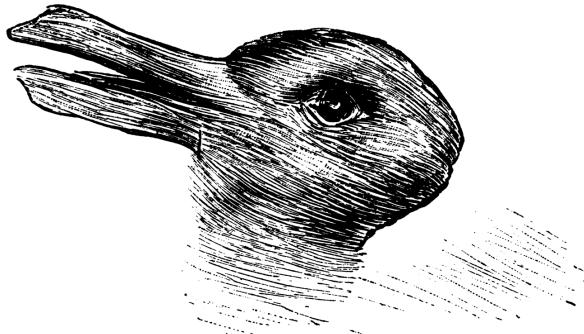


Figura 67: Red neuronal recurrente.

Welche Thiere gleichen einander am meisten?



Kaninchen und Ente.

anterior), obtenemos redes neuronales con memoria: las redes recurrentes [RNNs: Recurrent Neural Networks].

Las conexiones de una red recurrente pueden ser dirigidas, como sucede en las redes neuronales multicapa habituales, aunque también pueden ser bidireccionales. En este caso, obtenemos una red que es capaz de completar patrones incompletos, recibidos con ruido u ocultos parcialmente.

Un ejemplo clásico de este tipo de redes recurrentes con conexiones bidireccionales (esto es, enlaces no dirigidos) son las redes de Hopfield. Popularizadas por el físico norteamericano John Hopfield, del Instituto Tecnológico de California, son redes neuronales recurrentes con un comportamiento dinámico estable.

Igual que otros tipos de memorias asociativas, las redes de Hopfield son capaces de almacenar datos y de recuperarlos incluso cuando se les proporciona una entrada con ruido. Esta señal de entrada se puede interpretar como una versión corrompida de los datos almacenados en la red. La red se encarga de restaurar esos datos y generar, en su salida, el patrón original completo correspondiente a una entrada incompleta.

Las redes de Hopfield no se entrena en sentido estricto, sino que se diseñan para resolver problemas específicos.¹³⁷ Por ejemplo, se han utilizado para resolver problemas de optimización con restricciones, como el del viajante de comercio. Los pesos de la red se fijan para representar las restricciones del problema y la cantidad que se desea maximizar o minimizar. El comportamiento dinámico emergente de la red se encarga de proporcionarnos la solución al problema.

Las redes de Hopfield marcaron el inicio de una nueva generación de redes neuronales de tipo estocástico. Entre ellas se encuentran las

Figura 68: ¿Pato o conejo? Ilusión óptica publicada en octubre de 1892 por una revista satírica alemana, *Fliegende Blätter* (“hojas volando”, en alemán), con la siguiente leyenda en alemán: “¿Qué animales se parecen más entre ellos?”. Se hizo universalmente famosa porque el filósofo Ludwig Wittgenstein la incluyó para ilustrar su *Investigaciones filosóficas* en la descripción de dos formas de ver, “ver qué” y “ver como”. Fuente: Wikipedia.

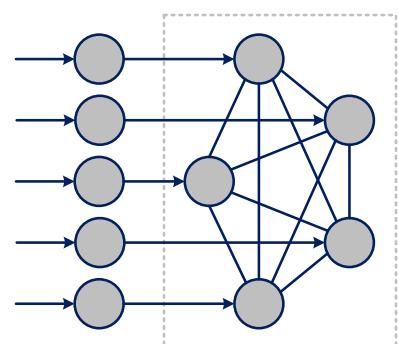


Figura 69: Red neuronal recurrente con conexiones bidireccionales.

¹³⁷ John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982. URL <http://www.pnas.org/content/79/8/2554>



máquinas de Boltzmann. Un tipo particular de máquinas de Boltzmann, las máquinas de Boltzmann restringidas o RBM [*Restricted Boltzmann Machines*], se emplean en la construcción de algunos modelos de redes neuronales utilizados en *deep learning*, como las “redes de creencia profunda” o DBN [*Deep Belief Networks*].

Este tipo de redes recurrentes se caracteriza por su comportamiento dinámico complejo. Ante una señal de entrada ambigua, que se pueda interpretar de distintas formas, una red recurrente puede oscilar entre varios estados semiestables. En cierto modo, la red es capaz de dudar. El fenómeno es similar al que experimentamos cuando vemos una imagen ambigua, con varias interpretaciones posibles incompatibles entre sí. Esas ilusiones visuales resultan muy atractivas: ¿pato o conejo?, ¿dos caras o una copa? Podemos interpretar fácilmente las imágenes de una forma u otra, aunque casi nadie puede ver ambas interpretaciones de forma simultánea. Las interpretaciones alternativas parecen alternarse en nuestra percepción. Desde el punto de vista de la dinámica de una red neuronal recurrente, cada interpretación corresponde a un atractor estable, con cierto grado de solapamiento entre los estados correspondientes a ambas interpretaciones, de ahí que pueda alternar.

Un fenómeno similar se observa cuando nos encontramos con determinadas formas geométricas bidimensionales con más de una posible interpretación tridimensional. Es el caso del cubo de Necker, ilusión óptica publicada por el cristalógrafo suizo Louis Albert Necker en 1832. El esqueleto del armazón de un cubo tridimensional no da pistas en cuanto a su orientación, por lo que se puede interpretar de distintas formas. El sistema visual humano selecciona una forma de interpretar la imagen ambigua para que ésta resulte consistente. Incluso podemos jugar con el cubo de Necker para construir una figura geométricamente imposible, del estilo del triángulo de Penrose. Este triángulo fue creado

Figura 70: ¿Dos caras frente a frente o una copa? Ilusión óptica diseñada en 1915 por el psicólogo danés Edgar Rubin e incluida en su tesis doctoral *Synsoplevede Figurer* (“figuras visuales”, en danés). Fuente: Wikipedia.

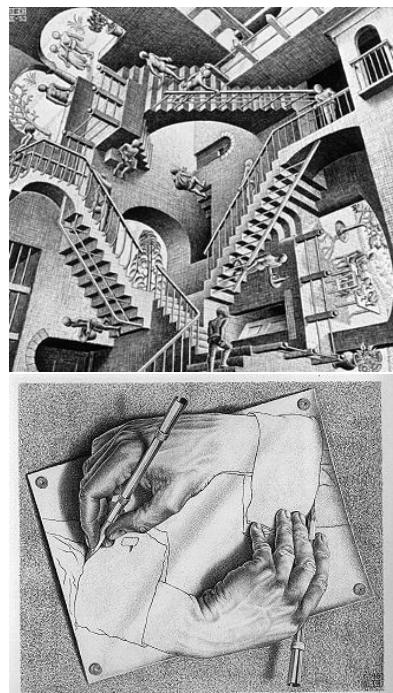
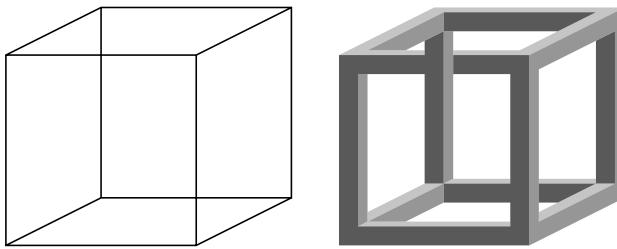


Figura 71: Ilusiones visuales de M.C. Escher: Relatividad (1953) y manos dibujando (1948).



por el artista sueco Oscar Reutersvärd en 1934 y popularizado por las famosas litografías del holandés Maurits Cornelis Escher, en las que abundan objetos imposibles y teselaciones.

Una diferencia notable entre las redes multicapa sin realimentación y las redes recurrentes es, evidentemente, cómo se reinician tras haberse activado. Las redes multicapa de tipo *feed-forward* se reinician inmediatamente después de cada activación, al carecer de memoria. Cada presentación de un nuevo patrón de entrada es independiente de la anterior, por lo que es como si estrenásemos una red nueva para cada uso. Las redes recurrentes, sin embargo, sí que tienen memoria. Por este motivo, corren el riesgo de saturarse y quedarse atascadas en un valor fijo si no las diseñamos cuidadosamente.

Simulación de neuronas biológicas

Los modelos de redes neuronales utilizados en neurociencia,¹³⁸ que intentan ser fieles al funcionamiento real de las neuronas biológicas, pueden utilizar otras estrategias. Algunos modelos, basados en *spikes*, resetean el estado de la neurona tras haber superado su umbral de activación, tal como sucede en una neurona biológica tras su activación y la generación de un *spike*. Otros modelos optan por modelar la frecuencia de activación de las neuronas, por lo que no simulan *spikes* individuales y, por tanto, no requieren reinicializar el estado de las neuronas.

Veamos, con algo más de detalle, en qué consisten esos modelos bioinspirados, que intentan reproducir la dinámica de las neuronas reales y que se emplean para simular las topologías, a menudo recurrentes, que se observan en redes neuronales biológicas:

- El modelo de respuesta de *spikes* [*SRM: Spike Response Model*] se rige por la siguiente ecuación para actualizar el potencial de la membrana de la neurona:¹³⁹

$$V_i(t) = \kappa(I_{ext}) + \eta(t - \hat{t}_i) + \sum_{j=1}^n w_{ij} \sum_{h=1}^H \epsilon_{ij}(t - \hat{t}_i, t - t_j^h)$$

donde κ representa la historia de las entradas externas a la neurona, η modela el período de refracción de la neurona (siendo \hat{t}_i el instante

Figura 72: Cubo de Necker (izquierda) y un cubo geométricamente imposible (derecha). Fuente: Wikipedia.

¹³⁸ Peter Dayan y L.F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press, 2001. ISBN 0262041995

¹³⁹ Wulfram Gerstner y Werner Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002. ISBN 0521890799

de tiempo de su última activación o *spike*) y el tercer término ϵ_{ij} modela las señales recibidas de otras neuronas, teniendo en cuenta sus últimos *spikes* (t_j^h), los retardos asociados a la transmisión de señales y la receptividad de la neurona en el instante t (en la que influye su período refractario, modelado con el término $t - \hat{t}_i$). Como podemos apreciar, se trata de un modelo algo más complejo que los usados habitualmente en redes neuronales artificiales.

- El modelo de Izhikevich¹⁴⁰ es más simple que el SRM que acabamos de describir y mucho más sencillo que modelar cada axón como una serie de circuitos RC en cascada, como en el modelo original de Hodgkin-Huxley. Es más abstracto y, por tanto, computacionalmente más económico si queremos simularlo en un ordenador. Además, es capaz de reproducir ciertos patrones de disparo observados experimentalmente en redes neuronales biológicas cuando se ajustan adecuadamente sus parámetros. Izhikevich modela dos características clave de cada neurona, el potencial eléctrico de su membrana (V_i) y un factor de recuperación (U_i) que emplea para representar el período de refracción de una neurona tras su activación:

$$\begin{aligned}\tau_m \frac{dV_i}{dt} &= 0.04V_i^2 + 5v + 140 - U_i + I_i \\ \tau_m \frac{dU_i}{dt} &= a(bV_i - U_i)\end{aligned}$$

donde τ_m es una constante de tiempo, I_i es la suma de todas las señales de entrada y las constantes “mágicas” que aparecen en la fórmula (los coeficientes numéricos) se ajustan experimentalmente usando datos de neuronas corticales. I_i se actualiza cada vez que una neurona aferente (de entrada) emite un pulso o *spike*, por lo que no hace falta mantener el historial de activaciones usado por el modelo SRM. Cuando V_i excede un umbral ($35mV$), se reinicia el estado de la neurona:

$$\begin{aligned}V_i &\leftarrow c \\ U_i &\leftarrow U_i + d\end{aligned}$$

En función de los parámetros a , b , c y d podemos obtener distintos tipos de patrones de disparo (p.ej. disparo regular, disparo por ráfagas o disparo periódico sin necesidad de estimulación externa).

- Los dos modelos anteriores modelan de forma explícita los pulsos o *spikes* individuales, por lo que han de simularse a una escala de tiempo muy pequeña (0.1ms en el caso del modelo de Izhikevich). Otros modelos, como el CTRNN [*Continuous-Time RNN*],¹⁴¹ modelan frecuencias de activación en lugar de activaciones individuales, lo que permite la realización de simulaciones más eficientes.

¹⁴⁰ Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, November 2003. ISSN 1045-9227. DOI: 10.1109/TNN.2003.820440

Otros modelos mediante los que se pueden modelar y reproducir multitud de patrones de disparo son AdEx [*Adaptive Exponential Integrate and Fire*] o GIF [*Generalized Integrate and Fire*].

¹⁴¹ Randall D. Beer y John C. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91–122, 1992. DOI: 10.1177/105971239200100105

En el modelo CTRNN empezamos por integrar las entradas:

$$s_i = \sum_{j=1}^n w_{ji}x_j + I_i$$

A continuación, modelamos la dinámica asociada al estado interno de la neurona, teniendo en cuenta la entrada neta s_i y un sesgo θ_i :

$$\frac{dV_i}{dt} = \frac{1}{\tau_i}[-V_i + s_i + \theta_i]$$

El modelo CTRNN utiliza la función logística para modelar la salida x_i de la neurona a partir de su estado interno V_i , usando un factor de ganancia g_i :

$$x_i = \frac{1}{1 + e^{-g_i V_i}}$$

Usando modelos como los anteriores, podemos simular redes neuronales complejas desde un punto de vista biológicamente plausible. No obstante, generalmente estaremos más interesados en crear redes neuronales que resuelvan problemas prácticos concretos. Para ello, tendremos que ajustar sus parámetros mediante un proceso de entrenamiento. Al entrenamiento de redes neuronales artificiales dedicaremos los próximos capítulos...

PARTE II. ENTRENAMIENTO DE REDES

En la que se estudian algoritmos para ajustar los parámetros de una red neuronal artificial durante su fase de entrenamiento, desde el aprendizaje del perceptrón hasta el algoritmo de propagación de errores (backpropagation), su uso en la práctica, las técnicas que se emplean para prevenir el sobreaprendizaje y las técnicas numéricas de optimización empleadas en deep learning.

Perceptrones

Aunque pueda parecer que el deep learning es el no va más en lo que se refiere a novedades de la I.A., en realidad, su base, como la de la Informática en general, comenzó a desarrollarse en plena Segunda Guerra Mundial. William McCulloch y Walter Pitts publicaron, en 1943, el primer modelo computacional de una red neuronal. No obstante, su modelo carecía de una funcionalidad básica: su capacidad de aprender.

No fue hasta más de una década después, en 1957, cuando Frank Rosenblatt publicó un artículo en el que se describía el primer modelo de red neuronal artificial capaz de aprender: el perceptrón. Su proyecto de investigación, apoyado por la Marina de EE.UU., causó cierto revuelo en 1958, cuando el New York Times informó del perceptrón como “el embrión de un ordenador electrónico que se espera que sea capaz de andar, hablar, ver, escribir, reproducirse a sí mismo y ser consciente de su existencia”. Una muestra más del optimismo desbordado con el que se recibieron los primeros (y muy rudimentarios) éxitos de la Inteligencia Artificial.

Otra década más tarde, en 1969, Marvin Minsky y Seymour Papert le dedicaron un libro al análisis de las capacidades reales del perceptrón, estudiando lo que realmente podían hacer y mostrando sus limitaciones. Aunque no fuese así, muchos interpretaron que esas limitaciones se extendían a todos los modelos de redes neuronales artificiales y esto dio lugar al invierno de la I.A. en lo que respecta a la investigación en redes neuronales artificiales, que no resurgiría con fuerza hasta mediados de los años 80. Aunque esa es otra historia que dejamos para más adelante.

La neurona de McCulloch y Pitts

Los orígenes de la Inteligencia Artificial y del deep learning se remontan a 1943. Ese año, dos investigadores del MIT, el psicólogo y neurofisiólogo Warren McCulloch y el lógico matemático Walter Pitts, describieron en un artículo el primer modelo computacional de una red neuronal.¹⁴² Su artículo serviría de base para el desarrollo posterior de las redes neuronales artificiales.

McCulloch y Pitts fueron los primeros investigadores que estudiaron el cerebro desde un punto de vista computacional. En vez de interpretar el funcionamiento de un ordenador como una secuencia de pasos predeterminados cuya ejecución secuencial permite resolver problemas, de forma meramente algorítmica, McCulloch y Pitts fueron capaces de intuir una forma alternativa de resolver problemas. Según ellos, un problema también se podía resolver utilizando dispositivos formados por una red de unidades interconectadas, neuronas según su terminología bioinspirada.

Aunque McCulloch y Pitts no desarrollaron ningún algoritmo de apren-

¹⁴² Warren S. McCulloch y Walter Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5(4): 115–133, 1943. ISSN 1522-9602. DOI: 10.1007/BF02478259

dizaje para sus redes neuronales que permitiese entrenarlas para resolver problemas concretos, su idea de interpretar el cerebro como un ordenador compuesto de múltiples unidades interconectadas daría lugar al desarrollo de las redes neuronales artificiales. Modelos posteriores permitirían que esas redes ajustasen sus patrones de interconexión mediante un mecanismo de realimentación basado en prueba y error. Cuando la red no proporcione la salida deseada, se ajustarán sus parámetros hasta que lo haga, de forma que sea capaz de aprender a realizar tareas de forma similar a como la plasticidad de nuestros cerebros biológicos nos permite aprender a realizar nuevas tareas.

Modelo computacional de McCulloch y Pitts

Ya en 1933, el psicólogo Edward Thorndike sugirió que el aprendizaje humano consistía en el fortalecimiento de alguna propiedad de las neuronas, entonces desconocida. El modelo computacional de McCulloch y Pitts realizaba una serie de suposiciones sobre el funcionamiento de las neuronas en una red neuronal:

- Neuronas binarias: La salida de las neuronas artificiales es binaria. Una neurona, o se activa, o no lo hace.
- Umbral de activación: Un número determinado de sinapsis, mayor que uno, debe activarse para la que neurona llegue a activarse. Ante la ausencia de estímulos externos, la neurona no se activa.
- Propagación de señales: La única demora que experimentan las señales de activación en su propagación se produce en las sinapsis.
- La actividad de una sinapsis inhibidora evita la excitación de la neurona. En el modelo original de McCulloch y Pitts, cualquier señal inhibidora de entrada impedía por completo la activación de la neurona.
- La estructura de la red de interconexión no cambia con el tiempo. Una suposición claramente falsa, ya que impide cualquier tipo de aprendizaje, aunque se consideró plausible cuando se propuso este modelo.

Utilizando estas suposiciones, se puede describir la acción de una red neuronal de McCulloch y Pitts utilizando lógica proposicional. Si representamos mediante $y_i(t)$ que una neurona está activa en el instante de tiempo t e indicamos que no lo está mediante $\bar{y}_i(t)$, podemos modelar cualquier red neuronal sin bucles de realimentación combinando únicamente cuatro expresiones:

- Precesión (una sinapsis conecta la salida de una neurona 1 con la entrada de una neurona 2):

$$y_2(t) = y_1(t - 1)$$

- Disyunción (dos sinapsis pueden activar la neurona postsináptica):

$$y_3(t) = y_1(t-1) \vee y_2(t-1)$$

- Conjunción (es necesario que dos sinapsis se activen para activar la neurona postsináptica):

$$y_3(t) = y_1(t-1) \wedge y_2(t-1)$$

- Negación (cuando una de las sinapsis es de carácter inhibitorio):

$$y_3(t) = y_1(t-1) \wedge \neg y_2(t-1)$$

McCulloch y Pitts pensaban que cada pulso o *spike* de una neurona venía a ser como el valor de verdad de una proposición en lógica proposicional, de forma que cada neurona combinaba valores de verdad para calcular el valor de otra proposición.

Aunque el modelo de McCulloch y Pitts no sea especialmente preciso a la hora de describir el funcionamiento real del cerebro, sentó las bases para el estudio de las neuronas como modelos computacionales simples que, combinados, son capaces de resolver problemas complejos. Un enfoque alternativo a la idea algorítmica de computación como serie ordenada de pasos que se ejecutan de forma lógica y completamente secuencial. Tengamos en cuenta que propusieron su modelo en 1943, seis años antes de que el psicólogo Donald Hebb sugiriese, en 1949, que el fortalecimiento de las conexiones entre las neuronas del cerebro es lo que da lugar al aprendizaje. Básicamente, McCulloch y Pitts trataron el cerebro como si fuese una máquina de Turing en la que cada neurona es un pequeño procesador digital simple y el cerebro en su conjunto es un mecanismo computacional.

Modelo neuronal de McCulloch y Pitts

El modelo neuronal propuesto por McCulloch y Pitts es un modelo sencillo de neuronas binarias con umbral [*binary threshold neurons*], en ocasiones denominadas unidades lineales con umbral o LTU [*linear threshold units*].

¿Cómo funcionan estas neuronas? En primer lugar, se calcula una suma ponderada de las entradas de la neurona, en la que cada entrada se multiplica por un peso sináptico:

$$z = \sum_i w_i x_i$$

o, en notación vectorial:

$$z = w \cdot x$$

A continuación, se genera como salida un pulso o *spike* de actividad cuando la suma ponderada de entradas excede un umbral preestablecido. Esto es, las neuronas del modelo propuesto por McCulloch y Pitts utilizan una función de activación binaria con un umbral θ :

$$y = \begin{cases} 1 & \text{si } z \geq \theta \\ 0 & \text{en otro caso} \end{cases}$$

Para simplificar un poco la notación, podemos añadir a todos los vectores de entrada un componente adicional fijo con valor 1. Asumiendo que dicho componente es la entrada $x_0 = +1$, dicha entrada tendrá asociada un peso w_0 que servirá para sesgar la respuesta de la neurona artificial en sustitución del umbral θ . De hecho, si establecemos $w_0 = -\theta$, podemos expresar la función de activación de la neurona de McCulloch y Pitts como:

$$y = \text{sgn}(z) = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{en otro caso} \end{cases}$$

Los ya familiarizados con el procesamiento digital de señales sabrán que dicha función de activación es la función escalón de Heaviside, también conocida como función escalón unitario, en honor al matemático, físico e ingeniero inglés Oliver Heaviside. Podemos simplificar la notación para describir las neuronas de McCulloch y Pitts si utilizamos $H(x)$ para la función escalón y la notación vectorial para representar el producto escalar de pesos y entradas:

$$y = H(z) = H(w \cdot x)$$

La función escalón de Heaviside H es una función discontinua cuyo valor es 0 para cualquier argumento negativo y 1 para cualquier argumento positivo. Dicha discontinuidad y, sobre todo, el hecho de que su derivada sea cero para todos los puntos en los que está definida, harán que no la utilicemos en la construcción de redes neuronales más complejas, aunque no adelantemos acontecimientos por el momento.

Es una práctica habitual sustituir el sesgo o *bias* de una neurona por un peso adicional asociado a una entrada fija. Al añadir la entrada adicional, $x_0 = +1$, el umbral de la neurona ha quedado reducido a un simple peso ($w_0 = -\theta$), que podemos tratar de forma uniforme junto con el resto de los pesos asociados a las demás entradas de la neurona. En el modelo de McCulloch y Pitts, un umbral positivo es equivalente a un sesgo negativo, mientras que un umbral negativo es equivalente a un sesgo positivo. En modelos posteriores, para los que sí existen algoritmos de entrenamiento, podremos aprender el umbral de activación de la neurona (su sesgo) como si fuese un peso sináptico más, sin tener que tratar de forma separada el umbral de activación de la neurona.

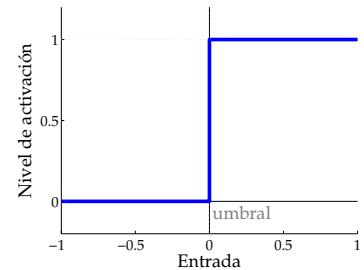


Figura 73: Función de activación de la neurona de McCulloch-Pitts.

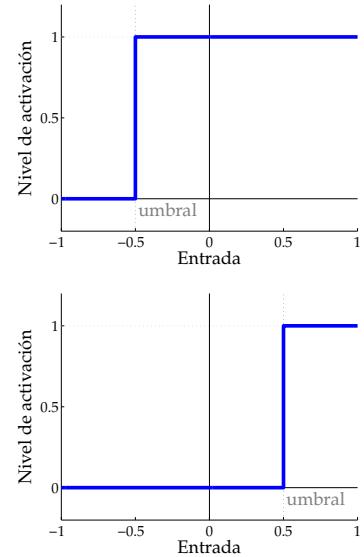


Figura 74: Efecto del umbral de activación sobre el funcionamiento de la neurona de McCulloch-Pitts.

En el caso del modelo de McCulloch y Pitts, para el que no propusieron algoritmo de entrenamiento alguno, seremos nosotros los que fijemos manualmente los parámetros de las neuronas que formen parte de una red neuronal. Aunque podríamos olvidarnos del umbral y representarlo en forma de peso w_0 , suele ser más sencillo razonar en términos de cuáles son los requisitos que deberían cumplir pesos y entradas con respecto al valor del umbral de activación para diseñar neuronas con la funcionalidad deseada.

La simulación del funcionamiento de una red neuronal utilizando el modelo de McCulloch y Pitts es una simple simulación de tiempo discreto: la activación de una neurona en el instante t depende de las salidas de las neuronas que la preceden en el instante $t - 1$.

Como las neuronas son binarias, el valor de activación de cada neurona se puede describir mediante una función lógica. El valor de activación de cada neurona (esto es, la función lógica que implementa) vendrá dado por el valor de sus entradas y los pesos sinápticos asociados a ellas, positivos en el caso de las sinapsis excitatorias y negativos en el caso de las sinapsis inhibitorias.

Si deseamos que una neurona se active cuando reciba k o más entradas excitatorias, sólo tenemos que establecer los pesos de cada una de sus entradas w y el umbral de activación de la neurona θ de tal forma que $kw \geq \theta > (k - 1)w$. Si lo que deseamos es conseguir una inhibición absoluta dada una señal inhibitoria en una neurona con n conexiones excitatorias de peso w y umbral de activación θ , sólo tenemos que establecer el peso $-h$ asociado a la entrada inhibitoria de tal forma que se verifique la siguiente condición: $\theta > nw - h$.

Estableciendo manualmente pesos y umbrales de activación, podemos modelar cualquier fenómeno que se pueda describir mediante una función lógica. Veamos cómo podríamos implementar fácilmente las expresiones básicas que nos permiten construir cualquier red neuronal sin bucles de alimentación:

- **Precesión** (una sinapsis conecta la salida de una neurona 1 con la entrada de una neurona 2, $y_2(t) = y_1(t - 1)$): Basta con conectar la salida de la primera neurona con la entrada de la segunda usando un peso positivo y utilizar un umbral de activación $\theta = 0$.
- **Disyunción** (dos sinapsis pueden activar la neurona postsináptica, $y_3(t) = y_1(t - 1) \vee y_2(t - 1)$): De nuevo, nos basta con conectar las salidas de las dos neuronas presinápticas con un peso positivo y utilizar un umbral de activación $\theta = 0$.
- **Conjunción** (es necesario que dos sinapsis se activen para activar la neurona postsináptica: $y_3(t) = y_1(t - 1) \wedge y_2(t - 1)$): En este caso, utilizamos pesos sinápticos positivos w para las entradas y establecemos un umbral de activación $2w \geq \theta > w$.

- Negación (cuando una de las sinapsis es de carácter inhibitorio: $y_3(t) = y_1(t - 1) \wedge \neg y_2(t - 1)$): Recurrimos a un peso positivo w para la entrada excitatoria y a un peso negativo $-h$ para la señal inhibitoria, usando como umbral un valor $\theta > w - h$.

Ejemplo: Percepción fisiológica del calor y del frío

Ilustremos el diseño de redes neuronales de McCulloch y Pitts utilizando un sencillo modelo de la percepción fisiológica del calor y el frío. Como tal vez haya experimentado personalmente en alguna ocasión, es un hecho comprobado experimentalmente que un estímulo frío puede percibirse como calor temporalmente. Igual que una ilusión óptica puede engañar a nuestro sistema visual, un estímulo particular puede confundir a nuestro sentido del tacto. Cuando un estímulo frío se aplica sobre la piel durante un período corto de tiempo, se percibe como calor. Si el mismo estímulo se aplica durante más tiempo, entonces se percibe el frío.

El uso de un modelo de simulación discreto, en el que los pasos de tiempo son discretos, nos permite modelar este fenómeno fisiológico utilizando neuronas de McCulloch y Pitts. Si asumimos que la activación de dos neuronas particulares se corresponde a nuestra percepción del frío ($y_{frío}$) y del calor (y_{calor}), podemos modelar el comportamiento de dichas neuronas en función de los estímulos reales que se reciben (*frío* y *calor*):

$$y_{frío}(t) = frío(t - 2) \wedge frío(t - 1)$$

$$y_{calor}(t) = calor(t - 1) \vee (frío(t - 3) \wedge \neg frío(t - 2))$$

Para conectar los estímulos con nuestra percepción subjetiva de los mismos, emplearemos un par de neuronas auxiliares que introducirán el retardo asociado a nuestra percepción e implementarán la función lógica deseada, de forma que la red neuronal responda de la misma forma que nuestra percepción del calor y del frío:

$$y_{calor}(t) = calor(t - 1) \vee z_1(t - 1)$$

$$z_1(t - 1) = z_2(t - 2) \wedge \neg frío(t - 2)$$

$$z_2(t - 2) = frío(t - 3)$$

Puede comprobar, si así lo desea, que la red neuronal que hemos diseñado responde con una percepción de calor a un estímulo frío puntual (activo durante un intervalo de tiempo), con una percepción de frío a un estímulo frío más persistente (activo durante, al menos, dos intervalos de tiempo) y siempre responde con una percepción de calor cuando se aplica una fuente de calor como estímulo.

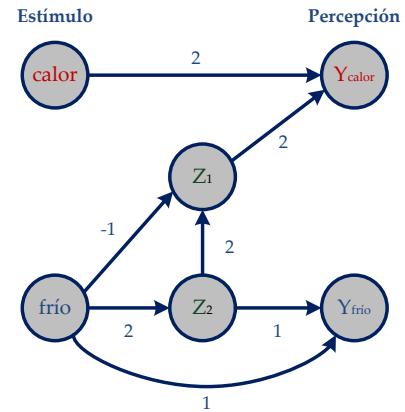


Figura 75: Red con neuronas de McCulloch y Pitts para modelar la percepción fisiológica del calor y el frío. Todas las neuronas de la red utilizan un umbral de activación $\theta = 2$.

Reconocimiento de patrones con redes neuronales

McCulloch y Pitts sentaron las bases que dieron lugar, cincuenta años después, al *deep learning*. Sin embargo, el modelo neuronal de McCulloch y Pitts carecía de un algoritmo de aprendizaje. No fue hasta unos años después cuando las redes neuronales artificiales comenzaron a ser capaces de aprender, con la popularización del algoritmo de aprendizaje del perceptrón de Frank Rosenblatt, epítome de la primera generación de redes neuronales artificiales.

Aparentemente, McCulloch y Pitts no se plantearon diseñar un algoritmo de aprendizaje que permitiese ajustar automáticamente los parámetros de una red neuronal formada por unidades lineales con umbral TLU. Sólo modelar el cerebro como si de una máquina de Turing se tratase. El propio Alan Turing, no obstante, describió el cerebro como un ordenador digital en una charla para la London Mathematical Society en 1947 y publicó un informe en 1948 en el que describía “máquinas no organizadas”.¹⁴³ Turing fue el primero en considerar el entrenamiento de redes neuronales, conectadas inicialmente de forma aleatoria. La aplicación de un mecanismo apropiado de “interferencia”, que emulase a la educación, les permitiría aprender a realizar cualquier tarea, dado el tiempo y el número de neuronas suficientes. De hecho, sugirió que el córtex de un bebé no es más que una “máquina no organizada” que se organiza mediante su entrenamiento, algo que Turing encontraba satisfactorio desde el punto de vista evolutivo y genético.

En los años 50 del siglo XX, distintos investigadores trabajaron en modelos computacionales basados en neuronas a ambos lados del Atlántico. En Estados Unidos, destacaron los trabajos de Belmont Farley y Wesley Clark, en el Laboratorio Lincoln del MIT; Marvin Minsky, durante la realización de su tesis doctoral en Princeton; y, especialmente, Frank Rosenblatt, en el Laboratorio Aeronáutico de Cornell. En Inglaterra, destaca William Ross Ashby, psiquiatra de Barnwood House, Gloucester, antes de trasladarse a la Universidad de Illinois en Urbana-Champaign en 1960. Otros pioneros menos conocidos fueron W.K. Taylor, del University College, Londres; Jack T. Allanson, de la Universidad de Birmingham; y los investigadores Raymond Louis Beurle y Albert M. Uttley del Radar Research Establishment, en Malvern, una pequeña ciudad del condado de Worcester, cerca de Birmingham.

Partiendo del modelo neuronal de McCulloch y Pitts, Belmont Farley y Wesley Clark, realizaron las primeras simulaciones informáticas de redes neuronales artificiales en 1954, con redes de hasta 128 neuronas que se entrenaban para reconocer patrones simples.¹⁴⁴ Su algoritmo de entrenamiento, o “modificador”, utilizaba un mecanismo de prueba y error que podríamos interpretar como una forma básica de aprendizaje por refuerzo: modificaba los pesos ajustables de una red neuronal arti-

¹⁴³ Alan M. Turing. Intelligent Machinery. Technical report, National Physical Laboratory, London, 1948. URL http://www.alanturing.net/intelligent_machinery/

Ross Ashby fue, junto con Warren McCulloch, Norbert Wiener, Alan Turing o W. Grey Walter, pionero en el campo de la cibernetica, definida por Wiener como el estudio científico del control y la comunicación en animales y máquinas.

¹⁴⁴ Belmont G. Farley y Wesley A. Clark. Simulation of self-organizing systems by digital computer. *Transactions of the IRE Professional Group on Information Theory*, 4(4):76–84, September 1954. ISSN 2168-2690. DOI: 10.1109/TIT.1954.1057468

ficial formada por unidades lineales con umbral LTU. Farley y Clark descubrieron, además, que la destrucción aleatoria de hasta el 10 % de las neuronas de una red neuronal entrenada no afecta significativamente a su rendimiento, una característica ya observada en el cerebro, que es tolerante a daños limitados causados por operaciones quirúrgicas, accidentes o enfermedades. De ahí, pasaron a estudiar cómo aprender a generalizar en el reconocimiento de patrones, utilizando lo que hoy llamaríamos aprendizaje supervisado.¹⁴⁵

En su tesis doctoral, publicada también en 1954, Marvin Minsky estudió modelos de aprendizaje por refuerzo y describió la construcción de una máquina analógica llamada SNARC [*Stochastic Neural-Analog Reinforcement Calculator*].¹⁴⁶ El proyecto SNARC, que arrancó en 1951 con financiación de las Fuerzas Aéreas de Estados Unidos, permitió construir físicamente una red neuronal que constaba de unas 40 sinapsis de tipo hebbiano. Esas sinapsis tienen una memoria en la que se almacena la probabilidad de que una señal llegue a su entrada y se propague a su salida (de ahí su carácter estocástico). Un operador puede indicar, pulsando un botón, cuándo proporcionar una recompensa a la red, lo que le permite ajustar automáticamente sus conexiones sinápticas mediante aprendizaje por refuerzo.

Pese a no ser el primero en pensar cómo se podría entrenar una red neuronal, fue el psicólogo Frank Rosenblatt el que más contribuyó a la popularización de la primera generación de redes neuronales artificiales: los perceptrones. Rosenblatt diseñó el algoritmo de aprendizaje del perceptrón y fue el primero en utilizar el término conexionista para distinguir su aproximación a la I.A. del enfoque simbólico. Los conexionistas, apelativo que pervive en la actualidad, equiparan el aprendizaje a la creación y modificación de conexiones sinápticas entre neuronas, frente al enfoque simbólico basado en lógica.

Rosenblatt se inspiró en el modelo de aprendizaje hebbiano, propuesto por Donald Hebb¹⁴⁷ en 1949 y desarrollado de forma independiente por el austriaco Friedrich Hayek¹⁴⁸ en la Universidad de Viena de los años 20, en su primer trabajo intelectual antes de dedicarse a sus estudios económicos, con los que acabaría obteniendo un premio Nobel en 1974 por su teoría del dinero y los ciclos económicos, base de la denominada Escuela Austríaca de Economía. Tanto Hayek como Hebb (y todos los conexionistas que les sucedieron) sitúan el aprendizaje a nivel sináptico, rechazando el asociacionismo simbólico de empíricos y positivistas lógicos.

Rosenblatt comenzó su investigación sobre redes neuronales artificiales en 1957 con su proyecto PARA [*Perceiving and Recognizing Automation*].¹⁴⁹ Aparentemente, desconocía el trabajo de Turing sobre “máquinas no organizadas”, igual que Donald Hebb cuando publicó su teoría de aprendizaje neuronal unos años antes. El mecanismo de aprendizaje de los perceptrones de Rosenblatt no era demasiado diferente a los propues-

¹⁴⁵ Wesley A. Clark y Belmont G. Farley. Generalization of pattern recognition in a self-organizing system. En *Proceedings of the Western Joint Computer Conference, March 1-3, 1955*, AFIPS '55 (Western), pages 86–91, 1955. doi: 10.1145/1455292.1455309

¹⁴⁶ Marvin Minsky. *Theory of Neural-Analog Reinforcement Systems and Its Application to the Brain-Model Problem*. PhD thesis, Princeton University, 1954. Más adelante, Minsky pondría en marcha el Laboratorio de Inteligencia Artificial del MIT, en 1970.

¹⁴⁷ Donald O. Hebb. *The Organization of Behavior*. John Wiley, New York, 1949

¹⁴⁸ Friedrich A. Hayek. *The Sensory Order: An Inquiry into the Foundations of Theoretical Psychology*. Routledge & Kegan Paul PLC, 1952. URL <https://archive.org/details/sensoryorderin00hay>

¹⁴⁹ Frank Rosenblatt. The perceptron: A perceiving and recognizing automaton. *Cornell Aeronautical Laboratory, Buffalo, New York*, Report 85-60-1, 1957. URL <https://goo.gl/17Fprk>

tos años antes por Farley y Clark en Estados Unidos, o Taylor, Uttley, Beurle y Allanson en Inglaterra. De hecho, el propio Rosenblatt reconoce que el mecanismo de generalización propuesto por Clark y Farley es esencialmente idéntico al encontrado en perceptrones simples.¹⁵⁰

Rosenblatt estudió las propiedades de los perceptrones realizando simulaciones por ordenador y analizando detalladamente sus propiedades matemáticas. Distinguía entre perceptrones simples con dos capas de neuronas (una de entrada y otra de salida) y perceptrones multicapa con tres o más capas. Rosenblatt generalizó el algoritmo de entrenamiento de Farley y Clark, que sólo servía para redes de dos capas, de forma que pudiese aplicarse a redes multicapa. De hecho, acuñó el término “corrección mediante propagación hacia atrás del error” [*back-propagating error correction*] que más tarde se adoptaría en la segunda generación de redes neuronales artificiales, que utilizaría un algoritmo de aprendizaje diferente llamado *backpropagation* o, por abreviar, *backprop*. Aunque el algoritmo original de Rosenblatt ya sólo se emplea para perceptrones simples, el algoritmo de propagación de errores se sigue utilizando en *deep learning*, que podríamos decir que corresponde a la tercera generación de redes neuronales artificiales.

En 1958, Rosenblatt viajó a Londres para presentar su perceptrón en un simposio del National Physical Laboratory. Allí, afirmó imprudentemente que su perceptrón era, junto al homeóstato¹⁵¹ de Ross Ashby, la única máquina capaz de mejorar “espontáneamente”, sin la tutela de un experimentador, ya que ignoraba los trabajos al respecto de Albert Uttley. Posteriormente, al publicar su descripción del perceptrón en una revista científica, reconoció las contribuciones de éste y las puso al mismo nivel que las de Hebb, Hayek y Ashby.¹⁵²

Aunque usualmente se atribuya a Rosenblatt la creación del primer algoritmo de entrenamiento de redes neuronales artificiales, hemos visto que las ideas no surgen espontáneamente de la nada, sino que en muchas ocasiones son el resultado natural de la evolución de las teorías científicas vigentes en un momento dado. En ocasiones excepcionales, se puede producir un cambio de paradigma [*paradigm shift*],¹⁵³ lo que llamaríamos un giro copernicano, pero habitualmente no es así. La ciencia avanza poco a poco, usando un mecanismo de prueba y error, con numerosos intentos independientes que, en ocasiones, se solapan y abordan un mismo problema desde perspectivas similares.

El modelo neuronal original de McCulloch y Pitts carecía de un algoritmo de aprendizaje, aunque los pesos sinápticos ya ofrecían el mecanismo que les permitiría aprender. Al ajustar dinámicamente los pesos asociados a las conexiones sinápticas entre neuronas, Clark, Farley, Minsky, Rosenblatt y Uttley consiguieron dotar a las redes neuronales de la capacidad de aprender. El perceptrón de Rosenblatt fue el modelo más popular en los años 60.

¹⁵⁰ Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962. URL <http://www.dtic.mil/docs/citations/AD0256582>

¹⁵¹ William Ross Ashby. *Design for a Brain*. Chapman and Hall, 1952. URL <https://archive.org/details/designforbrain00ashb>

¹⁵² Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. ISSN 1939-1471. DOI: 10.1037/h0042519. URL <http://psycnet.apa.org/journals/rev/65/6/386>

¹⁵³ Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 50th anniversary edition edition, 1962. ISBN 0226458113

Los perceptrones reciben ese nombre por el interés de Rosenblatt en aplicar modelos de redes neuronales artificiales a problemas relacionados con la percepción, como el reconocimiento de voz o el reconocimiento óptico de caracteres. En su esquema original, Rosenblatt propuso un modelo aproximado de la retina compuesto por tres capas: un área sensorial S [*sensory units*], un área asociativa A [*associator units*] con conexiones aleatorias fijas y localizadas (al estilo del campo de percepción visual de las neuronas de la retina) y un área de respuesta R [*response units*] en la que las neuronas compiten entre sí con conexiones inhibitorias (una competición del tipo *winner-takes-all*). El ajuste de los parámetros en este modelo simplificado de retina se realiza en una única capa, en las sinapsis de las unidades de respuesta, ya que las unidades sensoriales se limitan a proporcionar la entrada de la red y las unidades asociativas tienen conexiones fijas. De ahí que, normalmente, se interprete el perceptrón como una red neuronal de una sola capa.

Frank Rosenblatt pretendía ilustrar con su perceptrón algunas propiedades de los sistemas inteligentes. A diferencia del enfoque simbólico de McCulloch y Pitts, basado en lógica proposicional, Rosenblatt optó por un enfoque probabilístico para analizar un modelo idealizado de red neuronal: el fotoperceptrón que responde a patrones ópticos en la retina. Su proyecto de Cornell, financiado por la U.S. Office of Naval Research de la marina estadounidense, construyó un prototipo de red neuronal utilizando un ordenador con tarjetas perforadas, que en aquella época ocupaban una sala entera y tenían una capacidad de cálculo ridícula de acuerdo a los estándares actuales. Su primera implementación se realizó en software para un ordenador IBM 704 y, posteriormente, en hardware construido a medida: el “Mark I Perceptron” diseñado para reconocer imágenes. La máquina disponía de un array de 400 células fotoeléctricas de sulfuro de cadmio que formaban una matriz de 20 por 20. Estas células estaban aleatoriamente conectadas a las neuronas de la capa asociativa, mediante un cuadro de conexiones [*patchboard* o *switchboard*] que permitía jugar con diferentes configuraciones de la capa asociativa. Los pesos sinápticos ajustables del perceptrón se implementaban con potenciómetros (resistencias variables) cuyos valores se ajustaban durante el entrenamiento de la red con la ayuda de motores eléctricos.

En su configuración inicial, el perceptrón era incapaz de distinguir unos patrones de otros. Sin embargo, mediante un proceso de aprendizaje conseguía diferenciar patrones de interés. Este proceso de aprendizaje se realizaba por refuerzo: la actividad neuronal que contribuye a una respuesta correcta de la red neuronal se aumenta, mientras que la que no contribuye se disminuye. Después de decenas de iteraciones, el perceptrón aprendía a diferenciar tarjetas marcadas en su parte izquierda de tarjetas marcadas en su parte derecha. Una percepción muy rudimentaria que no impidió que el New York Times hablase de que la Marina “había revelado

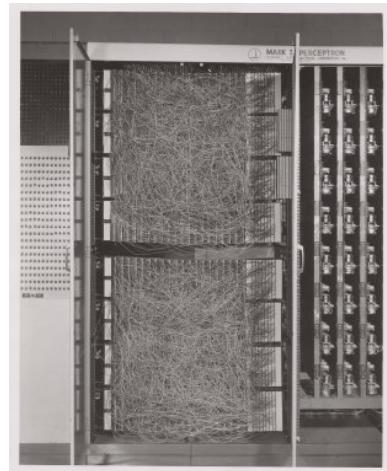


Figura 76: El Mark I Perceptron, la primera implementación en hardware del algoritmo de aprendizaje del perceptrón. A la izquierda, el cuadro de conexiones de la capa asociativa. A la derecha, los motores eléctricos y potenciómetros correspondientes a sus pesos adaptativos. Fuente: Wikipedia.

el embrión de un ordenador electrónico que se espera que sea capaz de andar, hablar, ver, escribir, reproducirse a sí mismo y ser consciente de su existencia".

Rosenblatt consiguió demostrar matemáticamente el teorema de convergencia del perceptrón. Si un perceptrón es capaz de discriminar correctamente entre dos clases, el algoritmo de aprendizaje garantiza que el perceptrón aprenderá a hacerlo en un número finito de pasos de entrenamiento. La clave está, como veremos más adelante, en establecer qué es lo que realmente puede aprender un perceptrón. Como ha sucedido con tantas técnicas de Inteligencia Artificial, unas expectativas exageradamente infladas a las que, en ocasiones, también contribuyen los propios proponentes de dichas técnicas sólo consiguen aumentar la decepción final cuando se descubren sus verdaderas limitaciones, que en este caso casi logran terminar con las redes neuronales artificiales como campo de investigación en Inteligencia Artificial.

Una vez dotadas de un algoritmo de aprendizaje, las redes neuronales artificiales son capaces de ajustar dinámicamente las conexiones sinápticas entre sus neuronas. Usualmente, este cambio se realiza de forma progresiva en función de la realimentación obtenida del éxito (o el fracaso) de la red durante su funcionamiento. Una red neuronal se ajusta automáticamente para producir la salida deseada, sin que su diseñador tenga que conocer cómo se puede resolver un problema ni la secuencia de pasos necesaria para llegar a una solución desde un punto de vista algorítmico. De hecho, ésa es la característica clave de los sistemas basados en redes neuronales artificiales: su capacidad para resolver problemas sin necesidad de conocer cómo se resuelven.

En cierta medida, las redes neuronales alcanzan la solución de un problema mediante un mecanismo de prueba y error. Este mecanismo les permite generar soluciones gradualmente mejores, reteniendo las conexiones entre neuronas que conducen a un rendimiento mejor. La arquitectura final de la red neuronal artificial es el resultado de su propia experiencia, de forma análoga a como se cree que se forman y evolucionan los circuitos neuronales del cerebro.

Aunque se suele utilizar la terminología propia del aprendizaje por refuerzo (p.ej. uso de recompensas y castigos) y partir de una interpretación biológica en la que el aprendizaje por refuerzo resulta esencial, algunos puristas defienden que el entrenamiento de redes neuronales artificiales no es más que un ejemplo de aprendizaje supervisado. A menudo, como acabamos de hacer en el párrafo anterior, se recurre al término "prueba y error" para hacer referencia a las redes que aprenden a partir de casos de entrenamiento (aprendizaje supervisado), de los que se deriva una señal de error que se emplea para ajustar sus pesos sinápticos. En aprendizaje por refuerzo, no obstante, se suele reservar el uso de "prueba y error" para la selección de acciones a partir de una señal de realimentación

Para otros, como el autor, el aprendizaje por refuerzo no es más que un tipo de aprendizaje supervisado en el que se emplea una forma particular de supervisión, más parecida a la proporcionada por un profesor que sigue el método so-crático.

en la que no se incluye información acerca de qué acción debería ser la correcta, algo que viene de serie con los ejemplos de entrenamiento.

El algoritmo de aprendizaje del perceptrón

Los perceptrones simples permiten reconocer patrones. Desde el punto de vista del aprendizaje automático, son clasificadores binarios: funciones que pueden decidir si una entrada dada pertenece a una clase específica o no. Para ser más precisos, los perceptrones son un tipo de clasificador lineal: clasificadores cuya frontera de decisión, la que separa los ejemplos de una clase de los de otra, viene dada en forma de línea recta en dos dimensiones, de plano en tres dimensiones o de hiperplano en general. Matemáticamente, la frontera de decisión definida por un clasificador lineal es de la forma:

$$b + \sum_i x_i w_i = 0$$

o, si consideramos el sesgo como un peso más, asociado a una entrada fija $x_0 = 1$:

$$\sum_i x_i w_i = 0$$

Por tanto, un perceptrón sólo será capaz de clasificar correctamente clases que sean linealmente separables.

¿Cómo se utiliza un perceptrón para clasificar? En primer lugar, se convierten los datos de entrada en un vector de características x . A continuación, se utilizan los pesos w asociados a cada una de esas características para obtener un valor escalar a partir del vector de entrada: $z = \sum_i x_i w_i$. Finalmente, este valor escalar z se emplea para clasificar el ejemplo x . Si z se halla por encima del umbral de activación de la neurona, ésta se activa y y se decide que el vector de entrada corresponde a un ejemplo de la clase objetivo, habitualmente denominada clase positiva ($y = 1$). Si no es así, el vector de entrada no provoca la activación de la neurona y el ejemplo se asocia a la clase negativa ($y = 0$).

¿Cómo se realiza el ajuste de los pesos del perceptrón? Básicamente, utilizando un algoritmo *online*, que procesa los ejemplos del conjunto de entrenamiento de uno en uno y va actualizando los pesos de forma incremental. En la práctica, se seleccionan los ejemplos del conjunto de entrenamiento utilizando cualquier política que garantice que todos los ejemplos de entrenamiento se acabarán escogiendo eventualmente. Para cada ejemplo del conjunto de entrenamiento:

- Si la salida del perceptrón es correcta, se dejan los pesos tal cual.
- Si la unidad de salida incorrectamente da un cero (un falso negativo), se añade el vector de entrada al vector de pesos.

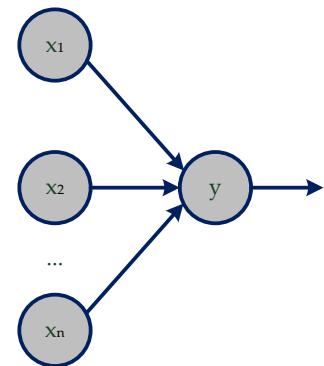


Figura 77: La arquitectura de la red neuronal artificial más sencilla: el perceptrón.

- Si la unidad de salida incorrectamente da un uno (un falso positivo), se resta el vector de entrada del vector de pesos.

El signo del error cometido nos indica en qué sentido hay que modificar los pesos. Habitualmente, recorreremos varias veces el conjunto de entrenamiento hasta que el algoritmo converja, la tasa de error baje por debajo de un umbral establecido por el usuario o no apreciemos una mejora significativa en la precisión del clasificador tras un recorrido completo. La versión más sencilla del aprendizaje consistiría simplemente en recorrer el conjunto de datos de entrenamiento un número fijo de veces. A cada recorrido del conjunto de datos durante el entrenamiento de una red neuronal se le suele denominar época. Éste sería el pseudocódigo del algoritmo de aprendizaje en su versión más simple:

```
for epoch=1:EPOCHS
    for i=1:dataset.size()
        data = dataset.getData(i);
        desired = dataset.getLabel(i);
        output = perceptron.getOutput(data);

        if (desired != output)
            delta = desired - output;
            perceptron.weights += delta*data;
```

El algoritmo de aprendizaje del perceptrón propuesto por Rosenblatt utilizaba entradas y salidas binarias, sin umbral. No obstante, es habitual utilizar una codificación bipolar de las entradas, $-1, +1$, ya que esta codificación tiene dos características que resultan útiles en la práctica. Por un lado, permite diferenciar situaciones en las que tenemos que trabajar con datos de entrada desconocidos, que podemos codificar utilizando un tercer valor $x = 0$. Por otro lado, la codificación bipolar usando -1 facilita el ajuste de los pesos durante el entrenamiento del perceptrón. Si utilizásemos una codificación binaria, sólo se modificarían los pesos asociados a las entradas distintas de cero.

Normalmente, para ajustar el sesgo o umbral de activación del perceptrón como si de un peso más se tratase, añadiremos una entrada adicional $x_0 = 1$ a los patrones de entrada, por lo que nuestros vectores de entrada serán de dimensión $m + 1$: $(1, x_1, x_2, \dots, x_m)$. El vector de pesos del perceptrón también será de tamaño $m + 1$: $(b, w_1, w_2, \dots, w_m)$. El primer peso será el sesgo o *bias* de la neurona, lo que nos permitirá aprender el umbral de activación del perceptrón fácilmente.

El algoritmo de entrenamiento del perceptrón comienza con la inicialización de sus pesos. Dado que el algoritmo no es particularmente sensible a los valores iniciales de los pesos, es habitual que se inicialicen a cero. También se pueden inicializar aleatoriamente (usando algún

Matemáticamente, el umbral de activación nos permite que el hiperplano que define la frontera de decisión del perceptrón no tenga que pasar por el origen, una característica necesaria para que nuestro clasificador sea siempre capaz de diferenciar dos clases linealmente separables.

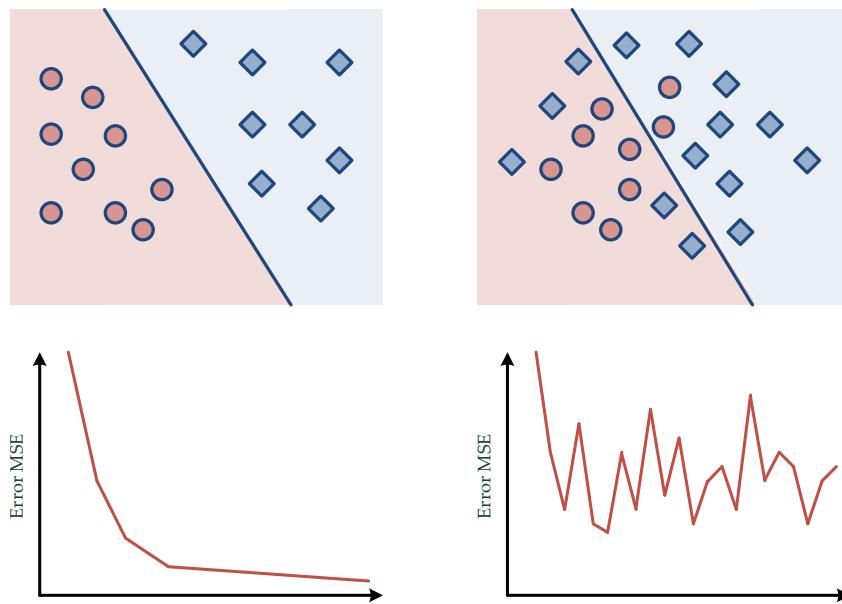


Figura 78: Entrenamiento del perceptrón para clases linealmente separables (izquierda) y para clases no linealmente separables (derecha).

valor aleatorio pequeño). A continuación, para cada ejemplo x se calcula la salida real del perceptrón y y se compara con la salida deseada d . Cuando ambas difieren, se actualizan los pesos usando la ecuación $w(t+1) = w(t) + (d - y)x$. Como, según nuestra convención, el sesgo de la neurona está asociado a una entrada fija $x_0 = 1$, su actualización siempre será de la forma: $w_0(t+1) = w_0(t) + (d - y)$. Es importante señalar que los pesos se actualizan tras mostrarle al perceptrón cada ejemplo de entrenamiento, sin esperar a que se haya completado una época completa.

Conforme avanza el aprendizaje, podemos evaluar la tasa de error del clasificador en cada época contabilizando el número de errores conocido, o bien calcular otras medidas de error para monitorizar la evolución del algoritmo de entrenamiento, como MAE o MSE:

$$MAE = \frac{1}{n} \sum_{j=1}^n |d_j - y_j|$$

$$MSE = \frac{1}{n} \sum_{j=1}^n (d_j - y_j)^2$$

Conforme vaya avanzando el entrenamiento, irá disminuyendo la tasa de error, el número de actualizaciones de los pesos del perceptrón será menor y, por tanto, su aprendizaje se irá haciendo más lento.

El algoritmo de aprendizaje del perceptrón garantiza encontrar un conjunto de pesos que proporcione la respuesta correcta, si tal conjunto existe. Dado que el perceptrón es un modelo de clasificación lineal, será

capaz de clasificar correctamente los ejemplos de entrada siempre que las clases sean linealmente separables, pero se mostrará incapaz de encontrar un conjunto de pesos adecuado para separar clases no linealmente separables. El algoritmo de aprendizaje del perceptrón converge si las clases son linealmente separables, pero nunca termina si las clases no son linealmente separables, ya que nunca llegará a un punto en el que el vector de pesos permita clasificar correctamente todos los ejemplos del conjunto de entrenamiento.

Si creamos una gráfica en la que se muestre la tasa de error, MAE o MSE para cada época de entrenamiento del perceptrón, observaremos que la medida de error será decreciente cuando las clases sean linealmente separables. Sin embargo, la medida de error que utilicemos puede oscilar si las clases de nuestro problema no son linealmente separables. La visualización gráfica del funcionamiento de nuestro clasificador nos puede servir para diferenciar ambos casos con facilidad.

Interpretación geométrica

Podemos interpretar geométricamente el algoritmo de entrenamiento del perceptrón si visualizamos los datos en un espacio de pesos en vez de en el espacio definido por las características de nuestros patrones de entrenamiento. En el espacio de pesos asociado a nuestro problema de aprendizaje:

- Cada peso del perceptrón define una dimensión. Si tenemos vectores de características de m dimensiones (x_1, x_2, \dots, x_m), nuestro espacio de pesos tendrá dimensión $m + 1$ al añadir el sesgo o umbral de activación del perceptrón.
- Cada punto del espacio de pesos representa un valor particular para el conjunto de pesos que determinan el comportamiento del perceptrón.
- Cada caso del conjunto de entrenamiento corresponde a un hiperplano que pasa por el origen $(0, 0, \dots, 0)$. Observe que, tras considerar el umbral de activación como un peso más, los hiperplanos asociados a los ejemplos de entrenamiento siempre pasan por el origen en nuestro espacio de pesos (el umbral era lo que nos permitía desplazar el hiperplano en el espacio de características y ahora es una dimensión adicional de nuestro espacio dual).

Mientras que en el espacio de características, cada ejemplo es un punto y el vector de pesos define un hiperplano (la frontera de clasificación del perceptrón), en el espacio de pesos cada vector de pesos es un punto y cada ejemplo define un hiperplano, que es perpendicular al vector de características del ejemplo. Se trata, simplemente, de una representación dual de nuestro problema de clasificación.

Veamos qué sucede exactamente con cada caso de nuestro conjunto de entrenamiento en el espacio dual de pesos. Evidentemente, el hiperplano asociado a un ejemplo divide en dos el espacio de pesos, por lo que el punto asociado a un vector de pesos concreto ha de quedar a uno de los lados del hiperplano:

- Cuando tenemos un ejemplo de la clase positiva ($y = 1$), el perceptrón clasificará correctamente el ejemplo siempre y cuando su vector de pesos corresponda a un punto situado en el lado del hiperplano al que apunta su vector de características.
- Cuando tenemos un ejemplo de la clase negativa ($y = 0$), el perceptrón clasificará correctamente el ejemplo sólo si su vector de pesos corresponde a un punto situado en el lado opuesto al que apunta su vector de características.

Como el perceptrón, básicamente, se limita a realizar un producto escalar para determinar su salida, el signo de este producto escalar debe corresponder a la clase del ejemplo del conjunto de entrenamiento. Si para un ejemplo de la clase positiva, el producto escalar del vector de entrada con el vector de pesos es negativo, la salida del perceptrón será la equivocada.

En otras palabras, cada ejemplo del conjunto de entrenamiento impone una restricción al conjunto de valores adecuado para los pesos del perceptrón. Estas restricciones vienen siempre dadas en forma de hiperplano que pasa por el origen.

Si queremos que el perceptrón clasifique correctamente todos los ejemplos del conjunto de entrenamiento, su vector de pesos deberá satisfacer simultáneamente todas las restricciones impuestas por los distintos ejemplos. El resultado, en caso de existir, será un “hipercono” de soluciones factibles cuyo ápice estará en el origen.

Dado que el hipercono de soluciones factibles forma una región convexa, la media de dos vectores de pesos adecuados será también un vector de pesos adecuado.

Para que el aprendizaje del perceptrón sea correcto, deberemos encontrar un punto que esté en el lado correcto de todos los hiperplanos definidos por los ejemplos del conjunto de entrenamiento. Ese punto, obviamente, puede no existir (cuando las clases no sean linealmente separables, obviamente). Ahora bien, si existe un vector de pesos que proporcione la respuesta adecuada para todos los casos de entrenamiento, el algoritmo de aprendizaje del perceptrón será capaz de encontrarla, como veremos a continuación.

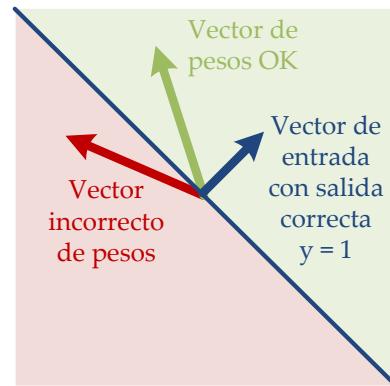


Figura 79: El hiperplano definido por un ejemplo de la clase positiva.

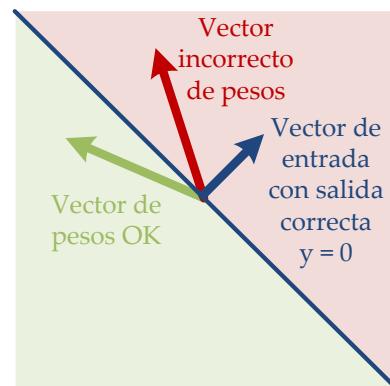


Figura 80: El hiperplano definido por un ejemplo de la clase negativa.

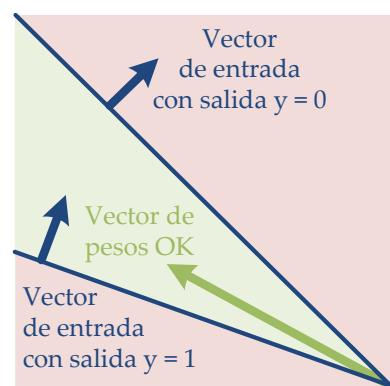


Figura 81: El hipercono de soluciones factibles del perceptrón.

Corrección del algoritmo de aprendizaje del perceptrón

El perceptrón es un clasificador lineal, por lo que su algoritmo de aprendizaje nunca llegará a un estado en el que todos los casos de entrenamiento se clasifiquen correctamente si el conjunto de entrenamiento no es linealmente separable. Esto es, el algoritmo no converge si los ejemplos de la clase positiva no pueden separarse de los ejemplos de la clase negativa por medio de un hiperplano. El algoritmo de entrenamiento del perceptrón, en este caso, no encontrará una solución “aproximada”, sino que oscilará de forma impredecible.

Sin embargo, cuando el conjunto de entrenamiento es linealmente separable, tenemos la garantía de que el algoritmo de aprendizaje del perceptrón convergerá, lo que nos permitirá encontrar un vector de pesos que clasifique correctamente todos los ejemplos de entrenamiento. Esta garantía viene dada en forma de cota superior al número de veces que el perceptrón debe ajustar sus pesos durante su entrenamiento hasta reducir su tasa de error a cero.

Para intentar demostrar la convergencia del algoritmo, podríamos partir, ingenuamente, de la hipótesis errónea de que, cada vez que el perceptrón comete un error, su algoritmo de aprendizaje acerca el vector de pesos actual hacia el hipercono de soluciones factibles. Sin embargo, esto no es siempre así.

Para verificar la convergencia del algoritmo de aprendizaje, hemos de considerar los vectores de pesos “generosamente factibles”. Estos vectores generosamente factibles quedan, dentro de la región factible, con un margen al menos tan grande como la longitud del vector de entrada. Cada vez que el perceptrón se equivoca, el cuadrado de la distancia a todos esos vectores de pesos generosamente factibles siempre se decrementa en, al menos, el cuadrado de la longitud del vector de actualización de los pesos. Por tanto, tras un número finito de errores, el vector de pesos deberá estar en la región factible, si ésta existe.

Supongamos que los vectores de entrada correspondientes a las dos clases linealmente separables de nuestro problema se pueden separar utilizando un hiperplano con un margen γ . Ese margen nos indica que existe un vector de pesos w , que consideramos normalizado ($\|w\| = 1$), de tal forma que $wx > \gamma$ para todos los ejemplos de la clase positiva y $wx < \gamma$ para todos los ejemplos de la clase negativa. Si M denota la norma máxima de los vectores de entrada, Block¹⁵⁴ y Novikoff¹⁵⁵ demostraron de forma independiente que el algoritmo de aprendizaje del perceptrón converge después de realizar $O(M^2/\gamma^2)$ actualizaciones de pesos.

La demostración se basa en la idea de que el vector de pesos se ajusta siempre en la dirección correspondiente a un producto escalar negativo. Veamos cómo demostrarlo siguiendo los pasos descritos por Laurene

¹⁵⁴ Henry David Block. The Perceptron: A Model for Brain Functioning. I. *Reviews of Modern Physics*, 34: 123–135, Jan 1962. doi: 10.1103/RevModPhys.34.123

¹⁵⁵ Albert B.J. Novikoff. On convergence proofs on perceptrons. En *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622, 1962

Fausett.¹⁵⁶

Partimos de nuestro conjunto de entrenamiento completo, que contiene dos subconjuntos de patrones, uno correspondiente a los ejemplos de la clase positiva P y otro correspondiente a los ejemplos de la clase negativa N . A partir de ellos, definimos un nuevo conjunto F en el que incluimos los ejemplos de la clase positiva y el negativo de los ejemplos de la clase negativa:

$$F = P \cup N^-$$

donde

$$N^- = \{-x | x \in N\}$$

La existencia de un vector de pesos w^* tal que

$$\begin{aligned} x \cdot w^* > 0 &\quad \text{si } x \in P \\ x \cdot w^* < 0 &\quad \text{si } x \in N \end{aligned}$$

es equivalente a que existe un vector de pesos w^* tal que

$$x \cdot w^* > 0 \quad \text{si } x \in F$$

Tal como hemos definido nuestro conjunto F , en el que hemos cambiado el signo de los ejemplos negativos incluidos en N , la salida correcta del perceptrón debe ser siempre $+1$ para todos los ejemplos de F . Cuando la respuesta del perceptrón sea incorrecta para un ejemplo $x \in F$, los pesos se actualizarán de acuerdo a

$$w(t+1) = w(t) + x$$

A continuación, comprobamos que la secuencia de conjuntos de vectores de entrada para los que se produce un cambio en los pesos es necesariamente finita:

- Cada vez que se produce un error de clasificación, se actualiza el vector de pesos:

$$w(1) = w(0) + x(0)$$

$$w(2) = w(1) + x(1) = w(0) + x(0) + x(1)$$

- Tras k actualizaciones:

$$w(k) = w(0) + x(0) + x(1) + \cdots + x(k-1)$$

Se puede demostrar que k no puede ser arbitrariamente grande. Sean w^* el vector de pesos tal que $xw^* > 0$ para todos los ejemplos de F y $m = \min\{xw^*\}$ para todos los ejemplos de F (ese mínimo existe dado que el conjunto de entrenamiento es finito). Entonces,

$$w(k) \cdot w^* = [w(0) + x(0) + x(1) + \cdots + x(k-1)] \cdot w^* \geq w(0)w^* + km$$

¹⁵⁶ Laurene V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice Hall, 1994. ISBN 0133341860

Utilizando la desigualdad de Cauchy-Schwartz, $(a \cdot b)^2 \leq \|a\|^2 \|b\|^2$, obtenemos:

$$\|w(k)\|^2 \geq \frac{(w(k) \cdot w^*)^2}{\|w^*\|^2} \geq \frac{(w(0)w^* + km)^2}{\|w^*\|^2}$$

Es decir, la longitud al cuadrado del vector de pesos crece más rápido que k^2 . Por otro lado, podemos demostrar que la longitud no puede crecer de forma indefinida, ya que

$$w(k) = w(k-1) + x(k-1)$$

y, dado que sólo se actualizan los pesos cuando el perceptrón se equivoca,

$$x(k-1)w(k-1) \leq 0$$

Teniendo en cuenta lo anterior, se obtiene que

$$\begin{aligned} \|w(k)\|^2 &= \|w(k-1) + x(k-1)\|^2 \\ &= \|w(k-1)\|^2 + 2w(k-1) \cdot x(k-1) + \|x(k-1)\|^2 \\ &\leq \|w(k-1)\|^2 + \|x(k-1)\|^2 \end{aligned}$$

Ahora bien, si llamamos $M = \max \|x\|$ para todos los ejemplos del conjunto de entrenamiento F , entonces

$$\begin{aligned} \|w(k)\|^2 &\leq \|w(k-1)\|^2 + \|x(k-1)\|^2 \\ &\leq \|w(k-2)\|^2 + \|x(k-2)\|^2 + \|x(k-1)\|^2 \\ &\leq \|w(0)\|^2 + \|x(0)\|^2 + \dots + \|x(k-1)\|^2 \\ &\leq \|w(0)\|^2 + kM^2 \end{aligned}$$

Combinando las dos desigualdades en las que aparecen m y M , obtenemos que el número de veces que los pesos se actualizan está acotado, por lo que el algoritmo de aprendizaje del perceptrón converge si las clases son linealmente separables:

$$\frac{(w(0)w^* + km)^2}{\|w^*\|^2} \leq \|w(k)\|^2 \leq \|w(0)\|^2 + kM^2$$

Para simplificar, podemos asumir que $w(0) = 0$, por lo que el número máximo k de actualizaciones de los pesos viene dado por

$$\frac{(km)^2}{\|w^*\|^2} \leq \|w(k)\|^2 \leq kM^2$$

Despejando k , obtenemos:

$$k \leq \frac{M^2}{m^2} \|w^*\|^2$$

Dado que w^* existe si las clases son linealmente separables y podemos asumir, sin pérdida de generalidad, que existe un vector solución de longitud unitaria, el número máximo de actualizaciones necesarias será, como mucho, M^2/m^2 .

Ahora bien, como en realidad desconocemos el vector de pesos solución w^* y, por tanto, también desconocemos m , que depende de w^* , no podemos predecir con exactitud el número de actualizaciones necesarias. Sólo podemos deducir que el algoritmo convergerá.

La demostración de convergencia nos muestra que el vector de pesos se ajusta siempre en la dirección que ocasiona un producto escalar negativo y se puede acotar la magnitud de ese ajuste, tanto por encima, $O(M/m)$, como por debajo, $O(M^2/m^2)$.

Además, teniendo en cuenta los resultados que hemos obtenido, podemos concluir lo siguiente:

- A la hora de garantizar la convergencia del algoritmo, no influye la codificación de los vectores de entrada, sea ésta binaria o bipolar. Sólo hace falta que su norma máxima M esté acotada.
- No influye, en lo que respecta a la convergencia del algoritmo, la inicialización de los pesos. Podemos inicializarlos directamente a cero, elegir un vector aleatorio de pesos iniciales o partir de un patrón cualquiera de los presentes en el conjunto de entrenamiento.

El algoritmo de aprendizaje del perceptrón va corrigiendo los errores conforme se los encuentra, desde cualquier configuración inicial de los pesos, hasta clasificar correctamente todos los ejemplos siempre que las clases sean linealmente separables.

Ahora bien, incluso aunque las clases sean linealmente separables, existe un número infinito de fronteras de decisión que nos podrían servir para separarlas. El algoritmo de aprendizaje del perceptrón, tal como lo hemos descrito, no utiliza criterio alguno para decidir con cuál de esas posibles soluciones se queda. Como consecuencia, puede que escoja una frontera de decisión que, aun clasificando correctamente todos los ejemplos del conjunto de entrenamiento, no resulte óptima para clasificar datos diferentes a los del conjunto de entrenamiento.

Las características y limitaciones del perceptrón, que hemos ido descubriendo conforme analizábamos la convergencia de su algoritmo de aprendizaje, nos permitirán diseñar variantes del perceptrón simple que, por ejemplo, garanticen su convergencia para problemas con clases no linealmente separables o nos seleccionen automáticamente la frontera de decisión óptima de entre todas las posibles.

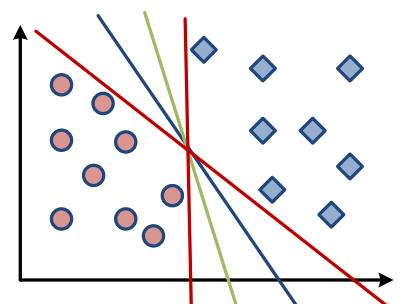


Figura 82: Múltiples fronteras de decisión permiten separar las dos clases y cualquiera de ellas podría obtenerse utilizando el algoritmo de aprendizaje del perceptrón, que no tiene forma de distinguir entre ellas.

Variantes y aplicaciones del perceptrón

El principal problema del perceptrón es que sólo sirve para clasificar correctamente cuando las clases son linealmente separables, que no suele ser lo habitual en la práctica. Ahora bien, si disponemos de n entradas binarias, siempre podríamos crear un problema de clasificación equivalente en el que las clases fuesen linealmente separables. Sólo tenemos que codificar por separado cada posible conjunto de entradas, por lo que nuestro perceptrón pasaría de n entradas a 2^n entradas, una por cada posible combinación de las n entradas originales. Utilizando esta arquitectura, las clases siempre serían linealmente separables, si bien el perceptrón resultante no resultaría especialmente útil. En primer lugar, por su prohibitivo coste, al tener un número exponencial de entradas. En segundo lugar, porque, aunque fuese capaz de clasificar correctamente los ejemplos del conjunto de entrenamiento, difícilmente sería capaz de generalizar correctamente. El mejor clasificador no es necesariamente el que clasifica perfectamente todos los casos de entrenamiento.

En el caso binario, el número de posibles combinaciones de entrada es 2^n , la cardinalidad del conjunto $\{0, 1\}^n$. Por tanto, el número de funciones binarias definidas sobre vectores binarios de n elementos será 2^{2^n} . Elegir una función particular de ese conjunto requerirá, por consiguiente, 2^n bits (o, si lo prefiere, $O(2^n)$ grados de libertad). De ahí que un perceptrón con 2^n sea un aproximador universal, pese a resultar poco práctico.

Si tenemos entradas continuas o analógicas, en vez de binarias, podríamos construir un perceptrón que incluyese una capa de preprocessamiento con unidades TLU y pesos aleatorios establecidos de antemano. Esa capa de preprocessamiento proyectaría nuestros patrones de entrada analógicos en un espacio binario. Dado el número suficiente de unidades en esa capa, los patrones serían linealmente separables. Es lo que hace el perceptrón α y la idea que subyace al diseño del área asociativa del perceptrón original de Rosenblatt.

Una forma alternativa de resolver problemas no lineales consiste en utilizar redes que incluyan conjuntos artificiales de características de entrada, sin llegar al extremo inviable del número exponencial de entradas que nos garantizaría la capacidad de aproximación universal (y la inutilidad del perceptrón a la hora de generalizar). Por ejemplo, podemos extender los vectores de entrada con las combinaciones resultantes de multiplicar pares de entradas $x_i x_j$. Es como si añadiésemos al perceptrón una capa inicial de unidades de multiplicación, de ahí que este tipo de redes reciba el nombre de perceptrones sigma-pi ($\Sigma-\Pi$).¹⁵⁷ Este esquema se podría extender para, además de las dependencias de segundo orden resultantes de las multiplicaciones $x_i x_j$, se tuviesen en cuenta efectos de orden n con productos del tipo $\prod x_{i_1} x_{i_2} \dots x_{i_n}$, con $n \geq 2$.

¹⁵⁷ Bartlett W. Mel y Christof Koch. Sigma-Pi Learning: On Radial Basis Functions and Cortical Associative Learning. En David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 474–481. Morgan Kaufmann, 1990. ISBN 155860-1007. URL <https://goo.gl/EQ2i8G>

Actualizaciones de los pesos

Existen otros algoritmos de aprendizaje automático que siguen una filosofía muy similar a la del perceptrón a la hora de construir un clasificador lineal. El algoritmo Winnow de Nick Littlestone,¹⁵⁸ en vez de realizar actualizaciones aditivas de los pesos, aplica actualizaciones multiplicativas. Al emplear un esquema multiplicativo, cuando se clasifica mal un ejemplo, los pesos se actualizan de la siguiente forma:

- Se promocionan los pesos $w_i(t+1) = \alpha w_i(t)$ cuando nos encontramos con un falso negativo.
- Se degradan los pesos $w_i(t+1) = w_i(t)/\alpha$ para los falsos positivos.

El parámetro α será estrictamente mayor que 1. Si le asignamos el valor 2 y utilizamos una codificación entera, esto nos permitiría realizar todas las operaciones de multiplicación a nivel de bits, ya que $2 * x \equiv x << 1$ y $x/2 \equiv x >> 1$, prescindiendo de esta forma del uso de multiplicaciones (y de las operaciones en coma flotante, más costosas que las operaciones sobre enteros).

El algoritmo Winnow, término que en castellano se traduciría por aventar (literalmente, hacer pasar una corriente de aire para separar el trigo de la paja), puede ser más rápido que el perceptrón cuando tenemos un número elevado de entradas y muchas de las entradas no son relevantes para el problema de clasificación. Sin embargo, si el número de entradas irrelevantes es bajo, el algoritmo Winnow puede ser más lento que el perceptrón.

Tasas de aprendizaje

En muchas técnicas de aprendizaje automático, se utiliza un parámetro adicional para controlar el tamaño de las actualizaciones que se realizan sobre los parámetros del modelo que se está entrenando. Este parámetro se suele denominar tasa de aprendizaje [*learning rate*] y denotar con la letra griega η (eta). La tasa de aprendizaje sirve para graduar la velocidad con la que se realizan modificaciones incrementales de los parámetros del modelo, de forma que $\Delta w = w(t+1) - w(t) = \eta \cdot error$, donde *error* es una señal de error que indica en qué sentido corregir los valores de los parámetros que deseamos ajustar.

El uso de tasas de aprendizaje es común en aprendizaje por refuerzo (*Q-learning*), en determinados tipos de ensambles (p.ej. *gradient boosting*) y en muchos tipos de redes neuronales artificiales, como las redes entrenadas con *backpropagation*. En general, se emplea siempre que utilicemos una función de error o pérdida diferenciable y utilicemos el gradiente de esa función de error para corregir los parámetros del modelo. Una estrategia que podemos utilizar también para resolver problemas de regresión lineal o logística.

¹⁵⁸ Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, April 1988. ISSN 0885-6125. DOI: 10.1023/A:1022869011914

A diferencia de otros algoritmos de aprendizaje que realizan modificaciones incrementales de los parámetros del modelo, el algoritmo de aprendizaje del perceptrón no requiere utilizar tasas de aprendizaje. No lo necesita porque multiplicar una actualización de los pesos por una constante simplemente reescalas los pesos pero no cambia el signo de la predicción realizada por el perceptrón. Pese a ello, en ocasiones se entrena un perceptrón utilizando tasas de aprendizaje:

- El caso tradicional de entrenamiento del perceptrón corresponde a utilizar una tasa de aprendizaje $\eta = 1$.
- Se puede emplear $\eta = 1/\|x\|$ para que el cambio en los pesos corresponda siempre a un vector unitario.
- Una tercera alternativa consiste en establecer $\eta = (x \cdot w)/\|x\|^2$, lo justo para que el patrón x pase a clasificarse correctamente en una única actualización de los pesos del perceptrón. Esta estrategia se conoce como versión de corrección absoluta de la regla del perceptrón.

La estabilidad del algoritmo no se ve afectada por el uso de una tasa de aprendizaje, si bien sí que puede afectar a su velocidad de convergencia (acelerándola con su versión de corrección absoluta, por ejemplo): convergerá cuando las clases sean linealmente separables y seguirá oscilando cuando no lo sean.

Zonas muertas

En su tesis doctoral de 1963, Crowell Hugh Mays, de la Universidad de Stanford, propuso extender el mecanismo de actualización de pesos del perceptrón de Rosenblatt añadiendo “zonas muertas” $\pm\gamma$ en torno a cero en las que los pesos se actualizan de forma diferente.¹⁵⁹ Si la magnitud de la entrada neta del perceptrón $z = w \cdot x$ es inferior al umbral γ , se actualizan los pesos del perceptrón independientemente de si su salida es correcta o no. Cuando $z \geq \gamma$, los pesos se adaptan de la forma habitual utilizando una versión normalizada de la regla original del perceptrón.

En particular, Mays propuso dos estrategias diferentes para la actualización de los pesos del perceptrón $\Delta w = w(t+1) - w(t)$:

- Su algoritmo de adaptación incremental [*incremental adaptation*], propuesto con anterioridad por Dave Block desde otra perspectiva,¹⁶⁰ realiza la actualización de los pesos de la siguiente forma:

$$\Delta w = \begin{cases} \eta(d(t) - y(t)) \frac{x(t)}{2\|x(t)\|_2^2} & \text{si } |z(t)| \geq \gamma \\ \eta d(t) \frac{x(t)}{\|x(t)\|_2^2} & \text{si } |z(t)| < \gamma \end{cases}$$

¹⁵⁹ Bernard Widrow y Michael A. Lehr. 30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. *Proceedings of the IEEE*, 78(9): 1415–1442, Sep 1990. ISSN 0018-9219. DOI: 10.1109/5.58323

¹⁶⁰ Henry David Block. The Perceptron: A Model for Brain Functioning. I. *Reviews of Modern Physics*, 34: 123–135, Jan 1962. DOI: 10.1103/RevModPhys.34.123

Cuando la zona muerta se elimina ($\gamma = 0$), el algoritmo de adaptación incremental se reduce a una versión normalizada de la regla de aprendizaje del perceptrón con tasa de aprendizaje η , asumiendo una codificación bipolar $\{-1, +1\}$ de las entradas del perceptrón (de ahí la división por 2).

Mays demostró que su algoritmo siempre converge y, además, normalmente funcionará mucho mejor que el algoritmo original del perceptrón cuando las clases no sean linealmente separables. La propuesta de Block y Mays aleja los pesos de cero cuando el perceptrón se sitúa en su zona muerta, reduciendo su sensibilidad a pequeños errores en los pesos. Seguirá oscilando cuando las clases no son linealmente separables, pero lo hará en una región con una tasa de error relativamente baja.

- La segunda propuesta de Mays es su algoritmo de relajación modificada [*modified relaxation*]:

$$\Delta w = \begin{cases} 0 & \text{si } |z| \geq \gamma \text{ y } d(t) = y(t) \\ \eta(d(t) - z(t)) \frac{x(t)}{\|x(t)\|_2^2} & \text{en otro caso} \end{cases}$$

Obsérvese que ahora se utiliza el error lineal $d(t) - z(t)$ en lugar del error cuantizado $d(t) - y(t)$ que se emplea habitualmente para ajustar los pesos del perceptrón, donde $y(t) = \text{sgn}(z(t))$. Cuando la salida del perceptrón es la correcta y queda fuera de la zona muerta, no se actualizan los pesos. Si comete un error o el perceptrón está en su zona muerta $\pm\gamma$, los pesos se modifican de acuerdo al producto escalar de pesos y entradas (la entrada neta de la neurona), en vez de utilizar la salida binaria de la misma, usando un ajuste por mínimos cuadrados [LMS: *Least Mean Squares*] que minimiza la función de error cuadrático $\text{error}(t) = (d(t) - z(t))^2$.

Esta regla de actualización de los pesos es similar a la regla utilizada por el ADALINE, otro de los modelos de redes neuronales de primera generación. Desarrollado por Bernard Widrow y el entonces estudiante de posgrado Marcian Edward “Ted” Hoff (el empleado número 12 de Intel, al que se considera inventor del microprocesador), el ADALINE estaba pensado para diseñar filtros adaptativos que facilitasen el procesamiento digital de señales (por ejemplo, para suprimir el eco en una conversación telefónica).¹⁶¹ El nombre del modelo de Widrow y Hoff proviene originalmente de *ADaptive LInear NEuron*; si bien, cuando decayó la popularidad de las redes neuronales a finales de los años 60, se rebautizó como *ADaptive LINear Element*, conservando su popularidad en el procesamiento de señales.

El algoritmo de relajación modificada de Mays equivale a la regla LMS del ADALINE cuando $\gamma \rightarrow \infty$. Para una zona muerta $0 < \gamma < 1$ y una tasa de aprendizaje $0 < \eta \leq 2$, el algoritmo converge para

¹⁶¹ Bernard Widrow y Marcian E. Hoff. Adaptive switching circuits. En *1960 IRE WESCON Convention Record, Part 4*, pages 96–104, New York, August 1960. Institute of Radio Engineers, Institute of Radio Engineers. URL <http://www-isl.stanford.edu/~widrow/papers/c1960adaptiveswitching.pdf>

clases linealmente separables. Cuando las clases no son linealmente separables, su funcionamiento es similar al algoritmo de adaptación incremental.

El perceptrón sigmoidal: Entradas y salidas continuas

El perceptrón original, que utilizaba unidades TLU y codificación bipolar o binaria, se puede extender al caso continuo utilizando una función de activación sigmoidal, como la tangente hiperbólica:

$$y(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{1 - e^{-2z}}{1 + e^{-2z}}$$

Para obtener cuál debería ser la regla de actualización de los pesos, podemos realizar un ajuste por mínimos cuadrados [LMS: Least Mean Squares] que minimice el error cuadrático dado por la función de pérdida instantánea, evaluada sobre el ejemplo actual:

$$J(w) = \frac{1}{2}(d(t) - y(t))^2$$

Para actualizar los pesos, utilizamos la dirección de máxima pendiente de la función de error:

$$w(t+1) = w(t) - \eta \nabla_w J(w)$$

lo que nos permite, usando un poco de aritmética, llegar a la expresión:

$$\nabla_w J(w) = -(d(t) - y(t)) \frac{dy(z)}{dz} [z(t)] x(t)$$

Si tenemos en cuenta cuál es la derivada de la tangente hiperbólica con respecto a la entrada neta z del perceptrón, $y'(z) = 1 - y^2(z)$, por lo que

$$\nabla_w J(w) = -(d(t) - y(t))(1 - y^2(t))x(t)$$

Por tanto, la actualización de los pesos de nuestro perceptrón sigmoidal se realizará de acuerdo a la siguiente regla de aprendizaje:

$$\Delta w = w(t+1) - w(t) = \eta(d(t) - y(t))(1 - y^2(t))x(t)$$

Esta regla de entrenamiento de los pesos de la red será la habitual cuando trabajemos con redes neuronales más complejas.

Si, por algún motivo, somos alérgicos a codificar valores reales en coma flotante, existe una alternativa curiosa para la codificación de entradas y salidas reales utilizando unidades bipolares. El denominado código del termómetro, término atribuido a Bernard Widrow. Según este código, un valor $x \in [0, 1]$ se representaría mediante k unidades binarias b_i cuyo valor vendría dado por:

$$b_i = \begin{cases} 1 & \text{si } x \geq i/(k+1) \\ -1 & \text{en otro caso} \end{cases}$$

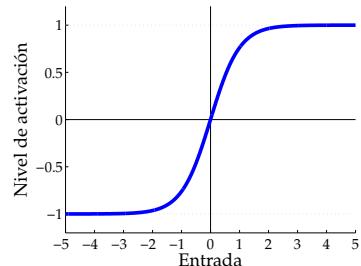


Figura 83: Función de activación sigmoidal bipolar: Tangente hiperbólica.

Por ejemplo, si $k = 4$ y $x = 0.6$, el valor se codificaría usando un vector $(+1, +1, +1, -1)$, el mismo código que se emplearía para cualquier valor entre 0.6 y 0.8, a partir del cual se utilizaría el vector $(+1, +1, +1, +1)$. Esto es, k unidades bipolares nos permiten distinguir $k + 1$ intervalos diferentes tras normalizar nuestros valores en el intervalo $[0, 1]$. No es la forma más eficiente de codificar un valor numérico en binario pero permite preservar la relación de orden entre valores consecutivos, algo que no conseguiríamos con una codificación binaria estándar, con la que un simple perceptrón binario sería incapaz de trabajar (salvo, claro está, que lo que pretendamos sea diferenciar entre los vértices del hipercubo definido por los bits con los que se codifica un valor real, algo harto improbable en la práctica).

El algoritmo del bolsillo: Clases no linealmente separables

Habitualmente, las clases de un problema de clasificación no son linealmente separables. Puede que nuestro conjunto de entrenamiento incluya ruido y errores, incluso ejemplos contradictorios. Simplemente, resulta imposible establecer un hiperplano que separe limpiamente las clases.

Cuando desconocemos si el conjunto de entrenamiento es linealmente separable, cualquiera de las versiones que hemos visto hasta ahora del algoritmo de aprendizaje del perceptrón puede oscilar de forma impredecible, sin ofrecernos garantías con respecto a la calidad del clasificador lineal que se termina obteniendo. El perceptrón no proporciona una solución “aproximada” a la que se vaya acercando paulatinamente, sino que puede llegar a aceptar cualquier solución, por dudosa que resulte su capacidad de generalización.

La solución más simple para intentar minimizar la varianza de los resultados obtenidos por el algoritmo tradicional de entrenamiento del perceptrón es el denominado algoritmo del bolsillo [*pocket algorithm*].¹⁶²

El algoritmo del bolsillo modifica el método estándar de aprendizaje del perceptrón introduciendo un mecanismo de realimentación positiva que estabiliza el aprendizaje cuando las clases no son linealmente separables. Simplemente, se limita a conservar en memoria (“en el bolsillo”) el mejor conjunto de pesos visto hasta el momento. Finalizado el proceso de entrenamiento, el algoritmo devuelve la solución del bolsillo en lugar del último conjunto de pesos del perceptrón, que puede distar mucho de ser el ideal.

Cuando un conjunto determinado de pesos clasifica correctamente una secuencia aleatoria de ejemplos de entrenamiento más larga que los pesos “del bolsillo”, el conjunto de pesos actual reemplaza a los del bolsillo. Este mecanismo de actualización, al que Gallant denomina trinquete, completa el nombre de su propuesta: algoritmo del bolsillo con trinquette

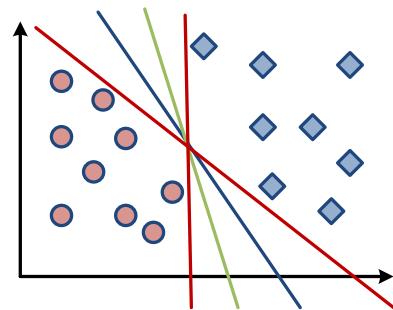


Figura 84: Múltiples fronteras de decisión permiten separar dos clases linealmente separables. Aun en este caso, que es el más sencillo, no sabemos cuál será la frontera elegida el algoritmo estándar de aprendizaje del perceptrón.

¹⁶² Stephen I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, June 1990. ISSN 1045-9227. doi: 10.1109/72.80230

[*pocket algorithm with ratchet*]. En cuanto se observa un conjunto de pesos mejor que el del bolsillo, se trinca.

Durante el entrenamiento del perceptrón, se le muestran aleatoriamente ejemplos del conjunto de entrenamiento. Conforme aumenta el número de iteraciones del algoritmo, los cambios en el bolsillo serán cada vez menos frecuentes. Stephen Gallant demostró formalmente que, dado un conjunto finito de ejemplos de entrenamiento y una probabilidad $p < 1$, existe un N tal que después de $n \geq N$ iteraciones, la probabilidad de que los pesos del bolsillo sean óptimos excede la probabilidad p . Experimentalmente, comprobó que el algoritmo del bolsillo comete un 20 % menos de errores en el conjunto de entrenamiento que los métodos estadísticos tradicionales de análisis discriminante lineal, tal como vienen implementados en herramientas como SPSS, cuando las clases no son linealmente separables.

Como sucede con el algoritmo estándar de aprendizaje del perceptrón, no existen mecanismos que nos indiquen de forma fiable cuántas iteraciones del algoritmo se deben ejecutar. Gallant recomienda comenzar con $i = 10000$ iteraciones y, si se actualizan los pesos del bolsillo pasadas $i/5$ iteraciones, ejecutar una nueva serie de iteraciones, con i incrementado un 50 % ($1.5i$). Este proceso se repite hasta que no se mejoren los pesos del bolsillo en las últimas $0.8i$ iteraciones de la última serie. El número de iteraciones necesario para obtener un conjunto de pesos óptimo puede resultar prohibitivo en la práctica, si bien se suelen obtener buenos pesos en un tiempo razonable. En cualquier caso, al tratarse de una red neuronal simple (la más simple que se puede diseñar), su tiempo de entrenamiento puede ser varios órdenes de magnitud menor que el necesario para ajustar los parámetros de otras redes neuronales más complejas, por lo que hay aplicaciones en las que puede resultar interesante su uso.

Cuando el conjunto de entrenamiento no sea linealmente separable, nuestro objetivo es minimizar el número de errores en el conjunto de entrenamiento, algo que el algoritmo del bolsillo nos ayuda a conseguir. No de forma gradual ni garantizada, pero sí de forma estocástica.

La idea de conservar, en un bolsillo, la mejor solución encontrada hasta el momento, es habitual en muchas técnicas de Inteligencia Artificial, incluso aunque sea “mejor” utilizando un indicador indirecto de la métrica que realmente nos interesa (la tasa de error en el conjunto de entrenamiento). Se le puede añadir bolsillo a otras variantes del perceptrón, como el algoritmo Winnow.¹⁶³ Se puede combinar el perceptrón y su bolsillo con otros modelos de clasificación, como los árboles de decisión, el modelo más popular de clasificación de la I.A. simbólica.¹⁶⁴ O, en otro contexto diferente, añadirle un “bolsillo” a un algoritmo genético para obtener un algoritmo genético elitista.

¹⁶³ Wray Buntine. Inductive knowledge acquisition and induction methodologies. *Knowledge-Based Systems*, 2(1): 52–61, March 1989. ISSN 0950-7051. DOI: 10.1016/0950-7051(89)90008-7

¹⁶⁴ Paul E. Utgoff. Perceptron trees: A case study in hybrid concept representations. En Howard E. Shrobe, Tom M. Mitchell, y Reid G. Smith, editores, *Proceedings of the 7th National Conference on Artificial Intelligence. St. Paul, MN, August 21-26, 1988.*, pages 601–606. AAAI Press / The MIT Press, 1988. ISBN 0-262-51055-3. URL <http://www.aaai.org/Papers/AAAI/1988/AAAI88-107.pdf>

El perceptrón de estabilidad óptima

Aunque el perceptrón garantiza su convergencia cuando las clases son linealmente separables, puede obtener una solución de calidad variable, tanto en el caso linealmente separable como en el que no lo es.

El perceptrón de estabilidad óptima, al que hoy se denomina máquina lineal de vectores de soporte (SVM lineal, para abreviar), se diseñó para resolver este problema. En problemas linealmente separables, el perceptrón de estabilidad óptima no sólo encontrará una solución que separe los ejemplos de una clase de los de otra, sino que escogerá el hiperplano que proporcione un margen de máxima separación entre las clases.

A lo largo de los años, se han propuesto diferentes algoritmos para maximizar la separación de la frontera de decisión del perceptrón con los ejemplos del conjunto de entrenamiento:

- Werner Krauth y Marc Mézard plantearon el problema en términos de un proceso iterativo de optimización. Su algoritmo utiliza la regla de aprendizaje del perceptrón para minimizar el solapamiento, de ahí que lo denominasen *Min-Over* [*minimum overlap*], y ofrece garantías con respecto a la estabilidad de la solución óptima.¹⁶⁵
- El *AdaTron* de J.K. Anlauf y Michael Biehl proporciona un método mucho más rápido que el algoritmo *Min-Over*, adaptando la regla de aprendizaje del ADALINE al perceptrón, aprovechando que el problema resultante de optimización cuadrática es convexo. De ahí su nombre: *AdaTron* = *ADAline percepTRON*.¹⁶⁶
- Andreas Wendemuth propuso una tercera versión “robusta” del algoritmo de aprendizaje del perceptrón. Su algoritmo, al que se suele denominar *Max-Over*, maximiza el margen de separación entre las clases incluso cuando ese margen es negativo (cuando las clases no son linealmente separables).¹⁶⁷

Los perceptrones de estabilidad óptima, en el caso linealmente separable, resuelven el problema devolviendo el hiperplano de mayor margen entre las clases. Para conjuntos de datos no linealmente separables, devuelven la solución que minimiza el número de ejemplos clasificados incorrectamente. En cualquier caso, lo logran sin memorizar estados previos (como el algoritmo del bolsillo) ni dar saltos estocásticos (como algunas metaheurísticas utilizadas en la resolución de problemas de optimización).

El perceptrón de estabilidad óptima, cuando se combina con el truco del kernel, da lugar a las máquinas de vectores de soporte o SVM [*Support Vector Machines*], una de las técnicas de aprendizaje automático más populares antes de la llegada del *deep learning*.

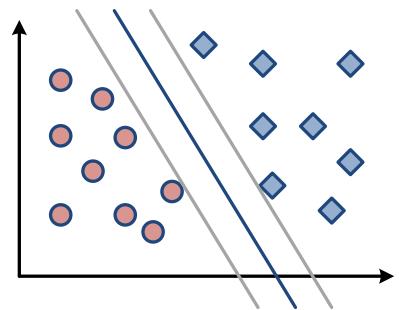


Figura 85: Maximización del margen en la frontera de decisión de un perceptrón.

¹⁶⁵ Werner Krauth y Marc Mezard. Learning algorithms with optimal stability in neural networks. *Journal of Physics A: Mathematical and General*, 20(11):L745, 1987. URL <http://stacks.iop.org/0305-4470/20/i=11/a=013>

¹⁶⁶ J. K. Anlauf y Michael Biehl. The AdaTron: An Adaptive Perceptron Algorithm. *EPL (Europhysics Letters)*, 10(7):687, 1989. URL <http://stacks.iop.org/0295-5075/10/i=7/a=014>

¹⁶⁷ Andreas Wendemuth. Learning the unlearnable. *Journal of Physics A: Mathematical and General*, 28(18):5423, 1995. URL <http://stacks.iop.org/0305-4470/28/i=18/a=030>

El perceptrón multiclasa

Las múltiples variantes del perceptrón que hemos visto hasta ahora nos permiten resolver problemas de clasificación binaria, en los que queremos diferenciar los ejemplos de una clase determinada (la clase positiva) de los de otra (la clase negativa). Extender el perceptrón a problemas de clasificación multiclasa es trivial. Simplemente, construimos k clasificadores binarios independientes, uno diferente para cada una de las clases del problema.

El perceptrón multiclasa, por tanto, no es más que un conjunto de k perceptrones independientes que se combinan para construir una única red neuronal artificial con k salidas. Como los distintos perceptrones son completamente independientes, se puede entrenar por separado cada uno de ellos (o en paralelo, si nuestro hardware nos lo permite).

La estrategia es la habitual para construir clasificadores multiclasa a partir de clasificadores binarios y se denomina *1 vs. all*. En este caso, la entrada del perceptrón multiclasa será la misma que para el perceptrón individual: un vector de características x . Su salida será un vector de bits de tamaño k en el que sólo un valor estará a 1 y todos los demás serán 0 (codificación *one-hot*).

Como si utilizamos unidades binarias, tipo TLU, corremos el riesgo de que se activen múltiples salidas y no sepamos cuál escoger, resulta habitual recurrir a unidades sigmoidales (o incluso, simplemente, lineales). Dado un ejemplo, éste se clasifica como perteneciente a la clase cuyo perceptrón alcanza un nivel de activación máximo. Esto es:

$$\hat{y} = \arg \max_j y_j = \arg \max_j x \cdot w_{y_j}$$

donde w_{y_j} corresponde al vector de pesos asociado a la clase y_j y hemos asumido que la función de activación del perceptrón $f(x \cdot w_{y_j})$ es estrictamente creciente, lo que nos permite prescindir de ella en la expresión anterior.

Una implementación simple del algoritmo de entrenamiento del perceptrón multiclasa podría tener este aspecto en pseudocódigo:

```
for epoch=1:EPOCHS
    for i=1:dataset.size()
        data = dataset.getData(i);
        label = dataset.getLabel(i);
        id = net.classify(data);

        if (id!=label)
            delta = desired[label] - net.getOutput();
            net.weights += learningRate*delta*data;
```

En este caso, recorremos el conjunto de datos un número determinado de veces (épocas, en la terminología habitual) y actualizamos los

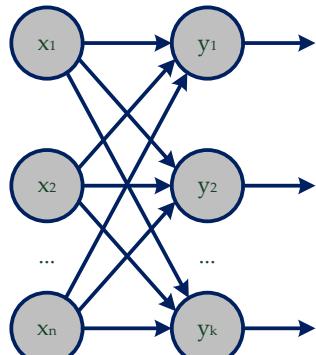


Figura 86: La arquitectura del perceptrón multiclasa: un perceptrón independiente para cada clase.

pesos cuando se produce un error de clasificación utilizando una tasa de aprendizaje dada. Las correcciones de pesos se pueden realizar utilizando la salida numérica de la red, de forma que se corrigen tanto los pesos correspondientes a la clase correcta (que se verán incrementados) como los correspondientes a las clases a las que no pertenece el ejemplo (que se verán decrementados). Esta estrategia de aprendizaje se reduce a la regla estándar del perceptrón cuando las salidas de la red pertenecen al conjunto $\{0, 1\}$ en codificación binaria o $\{-1, 1\}$ en codificación bipolar y la tasa de aprendizaje que se emplea es $\eta = 1$.

Como vimos al analizar la convergencia del algoritmo de aprendizaje del perceptrón, no es necesario inicializar los pesos. Podemos asumir que, inicialmente, la matriz de pesos W formada por los vectores de pesos de los k perceptrones es 0.

Por ejemplo, podemos aplicar un perceptrón multiclasa a la clasificación de dígitos manuscritos entrenando nuestro clasificador con la base de datos MNIST, que contiene varias decenas de miles de ejemplos (60k en el conjunto de entrenamiento y 10k en el conjunto de prueba).

Tras diez épocas de entrenamiento, obtendremos una tasa de error del 17.05 % en el conjunto de prueba. Tras cien épocas, la tasa de error en el conjunto de prueba bajará del 15 % (1489 errores, para ser precisos, un 14.89 %). Este valor sólo se consigue si utilizamos una tasa de aprendizaje adecuada, con un valor intermedio (0.05, 0.1, 0.2). Si la tasa de aprendizaje es demasiado baja (0.01, 0.02) o demasiado alta (0.5, 1.0), la tasa de error tras 100 épocas no conseguiremos que baje demasiado del 17 %. En cualquier algoritmo de aprendizaje, resulta esencial ajustar adecuadamente sus diferentes parámetros, incluso aunque su valor no tenga consecuencias formales desde el punto de vista teórico, como sucede con la tasa de aprendizaje del perceptrón.

Los resultados obtenidos con un perceptrón multiclasa, no obstante, no son especialmente buenos. Están muy alejados del estado del arte, tal como puede comprobar en <http://yann.lecun.com/exdb/mnist/>, donde, además de la base de datos MNIST, aparece la tasa de error conseguida por multitud de técnicas de aprendizaje automático.

¿Por qué se obtienen unos resultados tan mediocres con el perceptrón? Básicamente, porque un perceptrón multiclasa se limita a aprender un patrón único para cada clase del problema y, en problemas complejos, necesitaremos diferenciar los ejemplos de las distintas clases de una forma mucho más sutil. Más adelante, con una red multicapa, bajaremos del 2 % de error y, utilizando redes especializadas para trabajar con imágenes, como las redes convolutivas, lograremos bajar del 1 %.

Como algoritmo de aprendizaje del perceptrón multiclasa se puede utilizar cualquiera de los que hemos visto para ajustar los pesos de un perceptrón individual. En todos ellos, se van recorriendo los ejemplos del conjunto de entrenamiento, para los que se predice cuál debería ser



Figura 87: Algunos dígitos de la base de datos MNIST, utilizada como *benchmark* estándar en redes neuronales, la *drosophila* del aprendizaje automático, en palabras atribuidas a Geoffrey Hinton.

su clase, y se actualizan los pesos si la predicción no corresponde con la clase real del ejemplo de entrenamiento. Cuando la predicción es correcta, lo más habitual es dejar los pesos tal cual, sin cambios.

Si queremos minimizar la inestabilidad del algoritmo para clases no linealmente separables, podemos recurrir al algoritmo del bolsillo. También podemos hacer que la capa de salida sea competitiva, de tipo *winner-takes-all*, añadiendo conexiones inhibitorias entre las neuronas de salida (algo que implícitamente hicimos arriba).

En ocasiones, estas variantes del perceptrón reciben el nombre de máquinas lineales [*linear machines*], por motivos obvios. Las propiedades de convergencia de los algoritmos de aprendizaje del perceptrón para problemas de clasificación binaria se conservan para sus extensiones multiclase, ya que aprender los pesos de una máquina lineal con n entradas y k salidas a partir de t ejemplos de entrenamiento es equivalente a aprender los pesos de una sola neurona con kn entradas a partir de kt ejemplos de entrenamiento usando la construcción de Kesler.¹⁶⁸

El perceptrón promedio

El algoritmo de entrenamiento del perceptrón ajusta un vector de pesos de forma *online*, modificándolo cada vez que se le presenta un ejemplo de entrenamiento (y, usualmente, sólo cuando el perceptrón se equivoca en su predicción). Como vimos, el ajuste del vector de pesos se hace siempre de forma que se parezca más al vector de características del caso de entrenamiento, con el objetivo de que dicho vector quede en el lado correcto del hiperplano que define la frontera de decisión del perceptrón. Aunque el algoritmo converge y termina clasificando correctamente el conjunto de entrenamiento completo (cuando las clases son linealmente separables), la solución obtenida puede que no sea demasiado buena a la hora de generalizar.

A parte del perceptrón de estabilidad óptima, que maximiza el margen de separación de la frontera de decisión con los ejemplos del conjunto de entrenamiento, otra estrategia que puede ayudar a que el perceptrón generalice mejor consiste en promediar los pesos del perceptrón [*weight averaging*.¹⁶⁹ Esta modificación del algoritmo estándar consiste en hacer que el vector de pesos final que se obtiene como resultado del entrenamiento sea la media de todos los vectores de pesos que se utilizaron conforme se entrenaba el perceptrón. Si $w_{t,i}$ es el vector de pesos obtenido tras procesar el i -ésimo ejemplo del conjunto de entrenamiento durante la época t , el vector de pesos final del perceptrón será $w_{avg} = \sum_{t,i} w_{t,i} / (NT)$, donde N es el tamaño del conjunto de entrenamiento y T el número de épocas del entrenamiento. Yoav Freund y Robert Schapire, de AT&T Labs, mostraron que este promedio conduce a una solución más estable con mejores expectativas de generalización. Intuitivamente, el promedio

¹⁶⁸ Richard O. Duda, Peter E. Hart, y David G. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000. ISBN 0471056693

¹⁶⁹ Yoav Freund y Robert E. Schapire. Large Margin Classification Using the Perceptron Algorithm. *Machine Learning*, 37(3):277–296, December 1999. ISSN 0885-6125. DOI: 10.1023/A:1007662407062

sirve para amortiguar las oscilaciones propias del entrenamiento del perceptrón, sin recurrir a estrategias más costosas de optimización como las empleadas por los perceptrones de estabilidad óptima.

El pseudocódigo del algoritmo de entrenamiento del perceptrón promedio, que sus autores denominaron *average perceptron*, sería el siguiente:

```

perceptron.weights = 0;
perceptron.sum = 0;

for epoch=1:EPOCHS
    for i=1:dataset.size()
        data = dataset.getData(i);
        desired = dataset.getLabel(i);
        output = perceptron.getOutput(data);

        if (desired != output)
            delta = desired - output;
            perceptron.weights += delta*data;

        perceptron.sum += perceptron.weights;

total = EPOCHS * dataset.size();
perceptron.weights = perceptron.sum / total;

```

El algoritmo, tal como aquí se ha especificado, nos sirve como aproximación al resultado que se obtendría si se van almacenando todos los vectores de pesos correspondientes a las iteraciones del algoritmo de aprendizaje del perceptrón y se realiza la predicción como

$$s(x) = \sum_{t=1}^T \sum_{i=1}^N \text{sgn}(w_{t,i} \cdot x)$$

$$y(x) = \text{sgn } s(x)$$

Aunque ésta era la propuesta original de Freund y Schapire en su artículo, denominada *voted perceptron*, su implementación resultaría extremadamente ineficiente en la práctica, al requerir el voto ponderado de todos los vectores de pesos utilizados durante el entrenamiento del perceptrón. La implementación propuesta aquí es mucho más eficiente, ya que no requiere almacenar en memoria todos los vectores de pesos, sólo el vector actual y la suma acumulada de todos ellos.

Usando la base de datos NIST, en una versión anterior a la actual MNIST, Freund y Schapire obtuvieron resultados competitivos, sobre un 8 % de error tanto en su versión con votos ponderados [*voted perceptron*] como en su versión promedio [*average perceptron*]. Con algo de ingeniería de características, al introducir términos polinómicos como en

MNIST, precisamente, significa *Modified NIST*. MNIST es un subconjunto preprocesado de conjuntos de datos recopilados previamente por el NIST [*National Institute of Standards and Technology*] en Estados Unidos.

los perceptrones Σ - Π , el perceptrón promedio conseguía bajar la tasa de error por debajo del 2 %, aunque en este caso desaparecían las diferencias con respecto al algoritmo estándar de entrenamiento del perceptrón (con términos polinómicos de orden 5, las clases son linealmente separables para este conjunto de datos, por lo que cualquier algoritmo de clasificación lineal obtendrá la misma solución). El perceptrón promedio resulta, incluso, competitivo con las máquinas de vectores de soporte SVM,¹⁷⁰ que requieren un algoritmo de entrenamiento más complejo y costoso computacionalmente: 1.1 % de error de la máquina SVM frente al 1.6 % de error del perceptrón promedio usando términos polinómicos de orden 4.

La implementación ingenua que hemos realizado del perceptrón promedio requiere sumar vectores tras procesar cada ejemplo de entrenamiento, algo que se puede evitar si esa operación sólo se realiza cuando realmente se modifican los pesos:¹⁷¹

```

perceptron.weights = 0;
perceptron.avg = 0;
c = 1;

for epoch=1:EPOCHS
    for i=1:dataset.size()
        data = dataset.getData(i);
        desired = dataset.getLabel(i);
        output = perceptron.getOutput(data);

        if (desired != output)
            delta = desired - output;
            perceptron.weights += delta*data;
            perceptron.avg += c*delta*data;

        c++;

total = EPOCHS * dataset.size();
perceptron.weights -= perceptron.avg / c;

```

Sólo cuando se clasifica incorrectamente un ejemplo se realiza una operación vectorial de actualización de los pesos promediados, que se actualizan de la misma forma que los pesos actuales del perceptrón utilizando un factor multiplicativo c (el número de ejemplos vistos hasta el momento). El ajuste final de los pesos del perceptrón consigue el efecto deseado calculando el mismo promedio que en nuestra implementación ingenua.

Claudio Gentile, de la Universidad de Milán, propuso posteriormente una versión de estabilidad óptima del perceptrón promedio, denominada

¹⁷⁰ Corinna Cortes y Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995. ISSN 0885-6125. DOI: 10.1023/A:1022627411411

¹⁷¹ Harold Charles Daumé III. *Practical Structured Learning Techniques for Natural Language Processing*. PhD thesis, University of Southern California, 2006. URL <http://www.umiacs.umd.edu/~hal/docs/daume06thesis.pdf>

ALMA_p [*Approximate Large Margin algorithm w.r.t. norm p*].¹⁷² ALMA₂ es también competitivo con las máquinas de vectores de soporte, siendo su tiempo de entrenamiento necesario mucho menor que el de una SVM.

¹⁷² Claudio Gentile. A new approximate maximal margin classification algorithm. *Journal of Machine Learning Research*, 2:213–242, 2001. URL <http://www.jmlr.org/papers/v2/gentile01a.html>

El perceptrón estructurado

El perceptrón se puede emplear en procesamiento del lenguaje natural, para tareas tales como la delimitación de sintagmas nominales [*NP chunking*, de *Noun Phrase*], el etiquetado gramatical [*POS tagging*, de Part-Of-Speech] en el que se asigna a cada palabra de un texto su categoría gramatical o, incluso, al análisis sintáctico [*parsing*].

Para poder utilizarlo en estos contextos, es necesario diseñar una extensión del perceptrón que se conoce con el nombre de perceptrón estructurado. Básicamente, el perceptrón estructurado es una variante del perceptrón multiclase en la que se realiza una suposición de Markov. Esta suposición se emplea para clasificar secuencias (como la secuencia de palabras que forma un texto) teniendo en cuenta que la clasificación del elemento actual de la secuencia depende únicamente de las decisiones tomada para los ejemplos anteriores de la secuencia. La entrada del perceptrón estructurado es, por tanto, una secuencia (p.ej. de palabras) en vez de un único ejemplo. De la misma forma, su salida será una secuencia de etiquetas, que se utilizan para clasificar cada uno de los elementos de la secuencia de entrada. En otras palabras, el perceptrón estructurado es un clasificador de secuencias.

Como siempre, el entrenamiento del perceptrón estructurado se rea-liza recorriendo múltiples veces el conjunto de entrenamiento, formado por ejemplos etiquetados (x_i, y_i). Además, utiliza una función auxiliar GEN(x) que enumera conjuntos de candidatos para una entrada x . Cada ejemplo del conjunto de entrenamiento se codifica mediante un vector de características $\Phi(x, y)$ que podemos interpretar como una representación estructurada del patrón de entrada usando d dimensiones (de ahí la denominación de este tipo de perceptrón). Obviamente, esa representación dará lugar a un perceptrón con un vector w con d pesos, uno por cada característica de las empleadas para representar los ejemplos de entrenamiento.

El algoritmo de entrenamiento del perceptrón funciona de la siguiente forma. Para cada ejemplo de entrenamiento (x_i, y_i), se calcula la salida del perceptrón z_i :

$$z_i = \arg \max_{z \in \text{GEN}(x_i)} w \cdot \Phi(x_i, z)$$

Si la salida obtenida no es la correcta, $z_i \neq y_i$, entonces se actualizan los pesos del perceptrón utilizando la expresión:

$$w(i+1) = w(i) + \Phi(x_i, y_i) - \Phi(x_i, z_i)$$

Básicamente, como en el algoritmo original de Rosenblatt. El secreto está, obviamente, en cómo definir el vector de características adecuado para nuestro problema. Veamos cómo lo hizo Michael Collins, de AT&T Labs, para varios problemas habituales en procesamiento del lenguaje natural [*NLP: Natural Language Processing*]:

- En el etiquetado gramatical [*POS tagging*], la función $\text{GEN}(x)$ traduce una secuencia de entrada $w_{[1:n]}$ de longitud n al conjunto de todas las secuencias de etiquetas $t_{[1:n]}$ de longitud n . Para evitar la explosión combinatoria del número de secuencias de etiquetas, se utiliza el algoritmo de Viterbi, que no es más que una forma de aplicar la técnica de programación dinámica a problemas de optimización sobre secuencias. En un perceptrón multiclasa, sólo tenemos que predecir una de las k clases del problema. En un perceptrón estructurado, hemos de obtener la mejor secuencia de etiquetas de las k^n posibles. Con el algoritmo de Viterbi, este cálculo lo podemos realizar en tiempo polinómico, encontrando cómo de probable es una etiqueta POS t_i para una palabra w_i y cómo de probable es que la etiqueta t_i aparezca después de la etiqueta t_{i-1} .

El vector de características $\Phi(x, y) = \Phi(w_{[1:n]}, t_{[1:n]})$ se construye a partir de vectores de características locales $\phi(h, t)$: $\Phi(w_{[1:n]}, t_{[1:n]}) = \sum_{i=1}^n \phi(h_i, t_i)$. Cada característica local sirve de indicador: 1 si la característica está presente y 0 si no lo está. Por tanto, los vectores de características globales no son más que frecuencias de las características evaluadas localmente.

¿Qué tipo de características se pueden utilizar localmente? Las características locales vienen caracterizadas por pares (historia, etiqueta), donde la historia define el contexto en el que se toma la decisión. En este caso, la historia h_i puede venir dada por una tupla $\langle t_{-1}, t_{-2}, w_{[1:n]}, i \rangle$, donde t_{-1} y t_{-2} son las dos etiquetas asignadas a las dos palabras anteriores, $w_{[1:n]}$ es la secuencia de palabras e i es el índice de la palabra que deseamos etiquetar. De esta forma, se pueden definir funciones locales $\phi(h, t)$ para parejas (palabra, etiqueta) o para trigramas de etiquetas $\langle t_{-1}, t_{-2}, t \rangle$ que aparecen en el conjunto de entrenamiento.

El algoritmo de aprendizaje del perceptrón tenderá a incrementar los pesos asociados a las características que no estaban presentes en la secuencia propuesta $z_{[1:n]}$ y disminuirá los pesos asociados a características incorrectas en la predicción $z_{[1:n]}$. Por ejemplo, un trígrama $\langle D, N, V \rangle$ (determinante, nombre, verbo) será mucho más frecuente que $\langle N, D, V \rangle$ en textos reales.

En sus experimentos, Collins consiguió una tasa de error inferior al 3 % en el etiquetado gramatical, mejor que la obtenida utilizando otros modelos habituales en la clasificación de secuencias para NLP, como

los modelos de máxima entropía [*MEM: Maximum Entropy Model*]. Además, el entrenamiento del perceptrón se conseguía con un número de épocas significativamente inferior al necesario para entrenar los modelos de máxima entropía. De la misma forma, el perceptrón estructurado también mejoraba los resultados de los modelos de máxima entropía para la segmentación de sintagmas nominales [*NP chunking*], con un F-score superior al 93%.¹⁷³

- Para el análisis sintáctico [*parsing*], Collins también propuso un algoritmo incremental basado en el uso del perceptrón.¹⁷⁴ Como en el caso del etiquetado gramatical, el algoritmo realiza el análisis sintáctico de izquierda a derecha sobre la secuencia de datos de entrada.

Su perceptrón incremental sustituye la generación exhaustiva de alternativas de la función $\text{GEN}(x)$, que en *POS tagging* se realizaba con el algoritmo de Viterbi, por una búsqueda dirigida [*beam search*]. La función $\text{GEN}(x)$ genera distintos árboles de análisis sintácticos para una entrada x . Para algunas representaciones, como las basadas en gramáticas libres de contexto, se podría utilizar programación dinámica para encontrar el máximo sin necesidad de enumerar todas las alternativas (p.ej. utilizando una versión probabilística del algoritmo de Earley). Sin embargo, el lenguaje natural no se modela correctamente usando sólo gramáticas libres de contexto, por lo que, en general, no se pueden encontrar soluciones eficientes basadas en programación dinámica. De ahí que se optase por la alternativa de utilizar criterios heurísticos, basados en realizar una búsqueda dirigida incremental: en cada etapa, se conservan sólo las K alternativas más prometedoras.

El análisis incremental se realiza trabajando con hipótesis del tipo $\langle x, t, i \rangle$, donde x es la frase analizada, t es un árbol (parcial o completo) de análisis sintáctico para la frase e i indica el número de palabras de la frase que ya han sido procesadas. Para una frase de longitud n , cada candidato devuelto por la función $\text{GEN}(x)$ será de la forma $\langle x, t, n \rangle$.

La búsqueda dirigida parte de la hipótesis inicial $\langle x, \emptyset, 0 \rangle$. A continuación, utilizamos una función $\text{ADV}(\langle x, t, i \rangle)$ que, dada una hipótesis $\langle x, t, i \rangle$, genera un conjunto de hipótesis candidatas $\langle x, t', i+1 \rangle$. De alguna forma, la función ADV consigue incorporar una palabra más al árbol t construyendo un nuevo árbol $t+1$. De esta forma, podemos construir el conjunto $\text{GEN}(x) = H_n(x)$, siendo $H_0(x) = \{\langle x, \emptyset, 0 \rangle\}$ y $H_i(x) = \cup_{h' \in H_{i-1}(x)} \text{ADV}(h')$.

Usando esta definición, el número de elementos del conjunto $H_n(x)$ podría crecer exponencialmente y resultar intratable, por lo que se introduce una nueva función $\text{FILTER}(H)$ cuya función es limitar el conjunto de hipótesis candidatas, quedándose sólo con aquéllas más prometedoras: las que corresponden a mayores valores de $w \cdot \Phi(h)$.

El resultado final es el algoritmo de búsqueda dirigida utilizado por

¹⁷³ Michael Collins. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. En *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*, EMNLP '02, pages 1–8. Association for Computational Linguistics, 2002. DOI: 10.3115/1118693.1118694

¹⁷⁴ Michael Collins y Brian Roark. Incremental Parsing with the Perceptron Algorithm. En Donia Scott, Walter Daelemans, y Marilyn A. Walker, editores, *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics, 21-26 July, 2004, Barcelona, Spain.*, pages 111–118. Association for Computational Linguistics, 2004. URL <http://aclweb.org/anthology/P/P04/P04-1015.pdf>

el perceptrón incremental:

$$\begin{aligned} F_0(x) &= \{\langle x, \emptyset, 0 \rangle\} \\ F_i(x) &= \text{FILTER}\left(\cup_{h' \in F_{i-1}(x)} \text{ADV}(h')\right) \\ \text{GEN}(x) &= H_n(x) \end{aligned}$$

El secreto del algoritmo, una vez más, está en su ingeniería de características, utilizada en este caso para representar árboles sintácticos en forma de vectores de características.

Además, se pueden aplicar algunas optimizaciones para evitar tener que recalcular varias veces lo mismo, como mantener una caché de hipótesis candidatas (al estilo de la memorización de los algoritmos de programación dinámica) o utilizar una regla de parada que finalice la exploración de candidatos en cuanto nos demos cuenta que el análisis parcial de una frase nunca nos permitirá conseguir el análisis sintáctico correcto de la misma (en cuanto se ha producido un error y ninguna de las hipótesis incluidas en la búsqueda dirigida incluye una solución potencialmente válida).

Experimentalmente, el perceptrón incremental de Collins consigue un F-score del 88.8 % sobre un *treebank* de artículos del Wall Street Journal.

En todos los casos descritos, Collins empleó un perceptrón promedio en sus experimentos, para dotar de mayor estabilidad al algoritmo de aprendizaje y conseguir mejores resultados que los que se conseguían con el algoritmo tradicional de entrenamiento del perceptrón.

El algoritmo de aprendizaje del perceptrón viene a ser, en redes neuronales artificiales, como el Naïve Bayes para los métodos probabilísticos. Se sigue utilizando en la resolución de muchos problemas de clasificación, especialmente en aquéllos en los que los vectores de características pueden contener millones de elementos y resulta esencial la eficiencia del algoritmo de entrenamiento.

Además, como hemos visto, existen numerosas conexiones entre los perceptrones y otros modelos de clasificación populares, como las máquinas de vectores de soporte a las que dieron lugar los perceptrones de estabilidad óptima.¹⁷⁵

Limitaciones del perceptrón

Hemos visto que el perceptrón no es más que un clasificador lineal. Su algoritmo de entrenamiento sólo nos garantiza clasificar un conjunto de datos si las clases de nuestro problema son linealmente separables. Algunas de sus extensiones se pueden emplear para garantizar propiedades adicionales, como que el algoritmo proporcione una buena solución cuando las clases no son linealmente separables (algoritmo del bolsillo), que

En NLP, un *treebank* es una base de datos en la que, junto al texto, se proporciona el árbol sintáctico correspondiente a cada una de sus frases, de ahí lo de “banco de árboles”, en el sentido de banco de datos, claro está.

¹⁷⁵ Ronan Collobert y Samy Bengio. Links between Perceptrons, MLPs and SVMs. En *Proceedings of the 21st International Conference on Machine Learning*, ICML '04, pages 23–, 2004. ISBN 1-58113-838-5. DOI: 10.1145/1015330.1015415

la frontera de decisión finalmente escogida tenga propiedades deseables (estabilidad óptima) o que generalice mejor (perceptrón promedio). Además, con algo de ingeniería, podemos aplicarlo en problemas no triviales (perceptrón estructurado) y, usando heurísticas, en espacios de búsqueda enormes (perceptrón incremental).

También vimos cómo, a partir de las características de entrada de nuestro problema, se pueden diseñar características adicionales para tener en cuenta efectos de orden n (perceptrón sigma-pi), que pueden ayudarnos a separar linealmente el conjunto de clases de nuestro problema. De hecho, también observamos cómo, si se pueden añadir todas las características que deseemos, los perceptrones pueden hacer cualquier cosa (formalmente, sirven de aproximadores universales). Esto es, usando una entrada para cada posible vector de entrada (2^n en el caso binario), se puede discriminar cualquier función booleana. Sin embargo, un perceptrón así no será demasiado útil, ya que, además de necesitar un conjunto de entrenamiento de orden exponencial, no generalizará bien fuera de ese conjunto de entrenamiento, que es precisamente para lo que construimos un clasificador.

Si las características de entrada utilizadas por el perceptrón vienen predeterminadas, no obstante, existen severas limitaciones en cuanto a lo que un perceptrón simple es capaz de aprender. El ejemplo más famoso de la incapacidad de un perceptrón a la hora de resolver problemas no linealmente separables es la función booleana XOR [*eXclusive OR*].

La función XOR

La función XOR es una de las funciones booleanas con las que uno se encuentra por primera vez cuando aprende a programar. Denotada por el símbolo \oplus en circuitos lógicos y el operador \wedge en lenguajes de programación como C, Java o Python, su tabla de verdad es la siguiente:

Entrada x_1	Entrada x_2	Salida y
0	0	0
0	1	1
1	0	1
1	1	0

Concatenando operaciones XOR, uno puede calcular rápidamente la paridad de una secuencia de bits: el resultado será 1 cuando tenemos un número impar de bits a uno, 0 cuando tenemos un número par.

Si usamos la tabla de verdad del operador XOR para entrenar un perceptrón, cada una de las entradas de la tabla define una inecuación para los pesos (w_1 y w_2) y el sesgo del perceptrón (θ):

Tabla 1: La tabla de verdad de la función XOR [*eXclusive OR*].

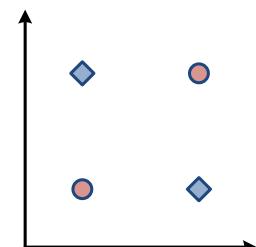


Figura 88: La función XOR no es linealmente separable.

$$\begin{aligned}
 w_1 + w_2 &< \theta \\
 w_1 &\geq \theta \\
 w_2 &\geq \theta \\
 0 &< \theta
 \end{aligned}$$

Este sistema de inecuaciones es imposible de resolver, por lo que el perceptrón nunca logrará aprender la función XOR. ¿Por qué? Porque la función XOR corresponde a un problema de clasificación en el que las clases no son separables. Los ejemplos positivos (salida 1) no pueden separarse con un plano de los ejemplos negativos (salida 0).

Geométricamente, podemos interpretar las restricciones impuestas por cada caso de entrenamiento (esto es, cada entrada de la tabla de verdad) si representamos gráficamente nuestro problema. Usando una dimensión para cada entrada, un vector de entrada corresponde a un punto en el plano, mientras que un vector de pesos define un hiperplano. Este hiperplano es perpendicular al vector de pesos y está a una distancia del origen dada por el umbral θ . Cada caso de entrenamiento impone una restricción al conjunto de soluciones factibles y es imposible encontrar una solución que satisfaga los cuatro casos simultáneamente.

La función XOR puede parecer excesivamente rebuscada. Uno podría pensar que este tipo de cosas no aparecen en conjuntos de datos reales. Sin embargo, si nos fijamos en su complementaria, apreciaremos mejor las limitaciones prácticas que posee un modelo neuronal tan simple como el perceptrón a la hora de reconocer patrones que aparecen a menudo en el mundo natural.

¿Son dos bits iguales?

Un perceptrón tampoco puede decidir si dos bits son iguales. La función de igualdad a nivel de bits (`==` o `eq` en la mayoría de lenguajes de programación), es la complementaria de la función XOR, por lo que también se la conoce como XNOR:

Entrada x_1	Entrada x_2	Salida y
0	0	1
0	1	0
1	0	0
1	1	1

Como sucedía con la función XOR, su función complementaria tampoco es linealmente separable, por lo que no seremos capaces de aprenderla con un perceptrón. Los cuatro casos de su tabla de verdad, nuevamente, definen cuatro inecuaciones imposibles de satisfacer simultáneamente:

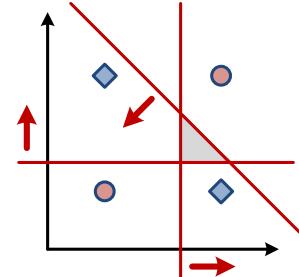


Figura 89: La función XOR define un sistema de inecuaciones lineales sin solución.

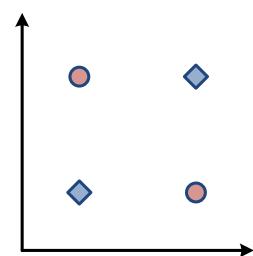


Figura 90: La función XNOR, que comprueba si dos bits son iguales, tampoco es linealmente separable.

Tabla 2: La tabla de verdad de la función de igualdad, o XNOR (el complemento de la función XOR).

$$\begin{aligned}
 w_1 + w_2 &\geq \theta \\
 w_1 &< \theta \\
 w_2 &< \theta \\
 0 &\geq \theta
 \end{aligned}$$

Geométricamente, podemos ver cómo los casos positivos (salida 1) tampoco pueden separarse en este caso de los casos negativos (salida 0) con la ayuda de un simple plano.

¿Tienen dos patrones el mismo número de bits?

Que el perceptrón sea incapaz de realizar una operación tan simple como comprobar si dos entradas son iguales tiene consecuencias a la hora de reconocer patrones que aparecen con frecuencia en el mundo real.

Supongamos que las características de entrada del perceptrón corresponden a píxeles de una imagen captada con la ayuda de algún tipo de cámara o sensor. Dado que un objeto puede encontrarse en diferentes posiciones con respecto al sensor que capta su presencia, la matriz de píxeles captada por el sensor, que se utiliza como entrada del perceptrón, puede aparecer desplazada de un caso a otro.

Sin embargo, un perceptrón será incapaz de discriminar entre patrones con el mismo número de píxeles si se admiten traslaciones. Pero, precisamente, éste es el tipo de situaciones para las que uno diseña un sistema de reconocimiento de patrones en primer lugar.

¿Por qué es incapaz el perceptrón de distinguir un patrón 1-3-1 de un patrón 2-1-2 cuando se admiten traslaciones?

- Para los patrones del primer tipo (1-3-1), podríamos entrenar el perceptrón usando el patrón en todas sus posiciones posibles. Al ir desplazando el patrón sobre nuestro array de sensores, cada uno de los píxeles se activará p veces, por lo que la entrada total recibida por el perceptrón será p veces la suma de sus pesos.
- Repitiendo el mismo proceso para el patrón del segundo tipo (2-1-2), entrenamos el perceptrón usando este segundo patrón en todas sus posiciones posibles. De nuevo, cada píxel se activará p veces, por lo que la suma total de las entradas recibidas por el perceptrón para los patrones 2-1-2 será, de nuevo, p veces la suma de sus pesos.
- Si queremos discriminar correctamente entre los dos patrones, los casos correspondientes a uno de ellos deberían proporcionar una entrada diferente a la proporcionada al perceptrón por los ejemplos correspondientes al otro patrón. Algo evidentemente imposible, ya que las entradas totales recibidas por el perceptrón son siempre las mismas para ambos tipos de patrones de bits.

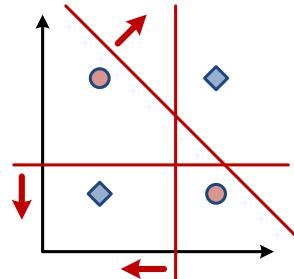


Figura 91: La función XNOR define un sistema de inecuaciones lineales sin solución.

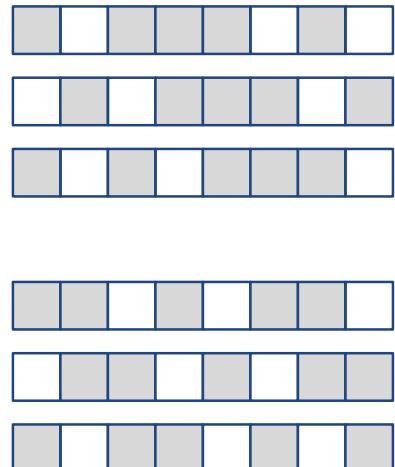


Figura 92: Dos patrones de bits que un perceptrón será incapaz de diferenciar: patrón 1-3-1 (arriba) y patrón 2-1-2 (abajo).

Por tanto, un perceptrón será incapaz de diferenciar patrones con el mismo número de bits si admitimos transformaciones tan habituales como una simple traslación en el espacio. Si queremos que nuestro perceptrón pueda reconocer este tipo de situaciones, tendremos que diseñar manualmente detectores de características que nos permitan discriminar correctamente. De esta forma, podríamos proporcionarle como entradas a un perceptrón aquellas características informativas correspondientes a subpatrones que puedan resultar útiles en cada problema particular (haciendo linealmente separables las clases de nuestro problema). Sin embargo, lo único que estaríamos haciendo es trasladar la parte más interesante del reconocimiento de patrones del algoritmo de aprendizaje automático a un proceso manual de ingeniería de características, que puede resultar arduo y costoso.

El libro de Minsky y Papert

Aunque, inicialmente, el perceptrón parecía prometedor (recordemos lo que llegó a publicar el New York Times en 1958), se demostró en poco tiempo que el perceptrón no podía entrenarse con éxito para resolver muchas tareas de reconocimiento de formas.

Marvin Minsky y Seymour Papert, del MIT, publicaron en 1969 un famoso libro en el que analizaban detalladamente las limitaciones del perceptrón.¹⁷⁶ Además del clásico ejemplo de la función XOR o el problema de la paridad de bits, Minsky y Papert revelaron otras limitaciones hasta entonces desconocidas del perceptrón, incluyendo el problema con el que ilustraron la peculiar portada de su libro (con una estética demasiado sesentera). Es uno de esos libros de los que todo el mundo habla pero que casi nadie ha leído realmente, como el Necronomicon de las historias de H.P. Lovecraft.

El libro también incluye demostraciones matemáticas, como la del teorema de invarianza de grupos [*group invariance theorem*]. Este teorema establece que un perceptrón nunca podrá aprender a diferenciar patrones que admitan transformaciones cuando las transformaciones a las que puedan estar sometidos dichos patrones forman un grupo, como en el caso del reconocimiento de patrones de bits con traslaciones. En resumen, la parte interesante del reconocimiento de patrones debe resolverse manualmente (p.ej. añadiendo nuevas características), pero no puede aprenderse utilizando un perceptrón.

La publicación del libro de Minsky y Papert, que investigaban en I.A. simbólica, supusieron un mazazo para la investigación en redes neuronales artificiales capitaneada por Frank Rosenblatt. Su detallado análisis matemático demostró limitaciones insalvables del perceptrón. Aunque los resultados de Minsky y Papert hacían referencia a perceptrones simples, exclusivamente, algunos los interpretaron como limitaciones intrínsecas

¹⁷⁶ Marvin Minsky y Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1st edition, 1969. ISBN 0262130432



Figura 93: Las portadas de las dos ediciones del libro de Minsky y Papert. En ambas se ilustra un problema de conectividad, imposible para el perceptrón y difícil también para nosotros. La edición expandida de 1987 (derecha) mejoró algo en su selección de colores de la portada con respecto a la edición original de 1969 (izquierda), al utilizar colores complementarios. Fuente: MIT Press. En matemáticas, un grupo es una estructura algebraica formada por un conjunto no vacío de elementos y dotada de una operación interna que combina cualquier par de elementos para componer un tercero dentro del mismo conjunto. Esta operación ha de satisfacer la propiedad asociativa, tener un elemento neutro (identidad) y poseer un elemento simétrico. Si, además, la operación es conmutativa, el grupo se denomina abeliano.

del modelo conexionista y de las redes neuronales en general. Algo a lo que también contribuyó que Minsky y Papert considerasen “estéril” el enfoque basado en redes neuronales multicapa, en este caso desde un punto de vista intuitivo, no formal, y meramente especulativo. En un informe de 1971,¹⁷⁷ Minsky y Papert insistieron en su desprecio hacia las redes neuronales indicando que un perceptrón Gamba, con capas ocultas, aunque no se conocía nada acerca de sus propiedades computacionales, no creían que fuese capaz de hacer mucho más que un perceptrón simple.

La interpretación incorrecta de los resultados derivados del análisis matemático del perceptrón y las críticas recibidas por los defensores de la I.A. simbólica hicieron que decayese el interés en redes neuronales artificiales, estancándose la investigación durante años y reduciéndose al mínimo la financiación de proyectos de investigación sobre el tema.

Tras la publicación del libro de Minsky y Papert, que coincidió en el tiempo con otras decepciones relacionadas con la Inteligencia Artificial, se entró en el período gris conocido como invierno de la I.A.. Las limitaciones, no sólo del perceptrón, sino también de los sistemas de reconocimiento de voz y traducción automática de la época, hicieron que la Inteligencia Artificial pasase temporalmente de moda y los esfuerzos se centrasen principalmente en la I.A. simbólica.

Aunque algunos investigadores siguieran trabajando en el tema, no fue hasta más de diez años después cuando las redes neuronales resurgieron con fuerza, a mediados de los años 80. Entonces fue cuando los investigadores en I.A. fueron realmente conscientes de que una red neuronal de tipo *feed-forward* con dos o más capas, incorrectamente llamada perceptrón multicapa [*MLP: multilayer perceptron*], era mucho más potente que un perceptrón simple, de una sola capa.

La I.A. simbólica se estaba estancando, la capacidad de cálculo de los ordenadores había mejorado y algunas publicaciones de James McClelland, David Rumelhart y su grupo de investigación PDP [*Parallel Distributed Processing*] demostraron el potencial de las redes neuronales. El caldo de cultivo perfecto para que surgiese con fuerza la segunda generación de redes neuronales artificiales, entrenadas utilizando el algoritmo de propagación de errores conocido por *backpropagation*.

Epílogo: Cómo resolver el problema XOR con perceptrones

Es importante ser conscientes de que los resultados matemáticos de Minsky y Papert se limitaban a los perceptrones simples, de una sola capa de pesos ajustables. Todas sus críticas hacia los perceptrones multicapa eran juicios de valor. Si queremos resolver el problema de la función XOR utilizando perceptrones sólo tenemos que añadir una capa adicional de perceptrones entre la entrada y la salida de la red. Esta capa intermedia, usualmente denominada capa oculta, es la encargada

Minsky, Papert, McCulloch, Pitts y Rosenblatt conocían perfectamente que, aunque un perceptrón simple no sea más que un humilde clasificador lineal, una red neuronal proporciona un modelo computacional equivalente a una máquina de Turing, capaz de calcular cualquier función booleana.

¹⁷⁷ Marvin Minsky y Seymour Papert. Artificial Intelligence: Progress Report. MIT, Artificial Intelligence Memo No. 252, 1972. URL <https://goo.gl/NH6vu5>

El libro de Minsky y Papert se reeditó en 1987 con una versión expandida, publicada tras la muerte de Rosenblatt y dedicada a su memoria. Minsky y Rosenblatt, pese a sus disputas académicas, se conocían desde la adolescencia, ya que ambos estudiaron, con un año de diferencia, en el mismo instituto del Bronx neoyorquino.

de extraer características de los vectores de entrada que hagan las clases linealmente separables.

En el caso de la función XOR con dos entradas y una salida, nos bastará con incluir una capa oculta con dos neuronas. Esas dos neuronas de la capa intermedia nos permitirán dividir el espacio de entrada en tres regiones diferentes, con lo que la neurona de la capa de salida podrá proporcionar la salida correcta de la función XOR.

Una primera posibilidad sería que una de las neuronas ocultas identificase el patrón de entrada 11 y la otra implementase la función OR. A continuación, la neurona de salida eliminaría el patrón 11 del resultado final usando un peso inhibitorio (negativo) para la neurona asociada al patrón 11:

Neurona	Entradas		Umbral θ
y_{00}	x_1	x_2	0.5
y_{11}	x_1	x_2	1.5
y_{xor}	$0.6y_{00}$	$-0.2y_{11}$	0.5

Un segunda posibilidad consistiría en utilizar conexiones inhibitorias en la capa oculta, para detectar por separado los patrones 01 y 10. La neurona de salida, simplemente, implementaría la función OR de las salidas de las neuronas ocultas:

Neurona	Entradas		Umbral θ
y_{10}	$2x_1$	$-1x_2$	2
y_{01}	$-1x_1$	$2x_2$	2
y_{xor}	$2y_{10}$	$2y_{01}$	2

Si, en lugar de neuronas binarias con umbral, utilizamos otro tipo de neuronas, con funciones de activación continuas, la misma topología de red nos permite resolver el problema XOR.

Por ejemplo, si usamos unidades lineales rectificadas (ReLU) resulta sencillo implementar la función XOR. En este caso, las dos unidades ocultas h_1 y h_2 pueden compartir los mismos pesos. Sólo tenemos que ajustar su sesgo: 0 para la primera neurona oculta, que se limita a contar el número de entradas con valor 1; -1 para la segunda, que sólo se activa si ambas entradas son 1. La neurona de salida calcula la diferencia entre los niveles de activación de las dos neuronas ocultas, anulando su salida en el momento en el que recibe una entrada positiva proveniente de h_2 (una solución equivalente a la primera que vimos usando perceptrones binarios en la capa oculta).

Éstas son sólo algunas de las muchas soluciones que se pueden encontrar manualmente para el problema XOR utilizando redes de diferentes tipos. De hecho, poco después de la publicación del libro de Minsky y Papert,

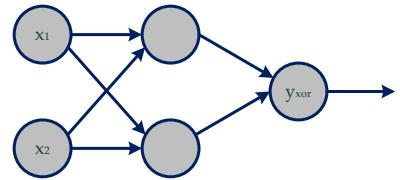


Figura 94: Solución del problema XOR utilizando tres neuronas y dos capas de perceptrones.

Tabla 3: Una red de perceptrones para resolver el problema XOR. De los tres patrones que activan la neurona y_{00} (01, 10 y 11), la neurona de salida excita el patrón 11, que es el único que activa la neurona y_{11} .

Tabla 4: Una solución alternativa basada en perceptrones para resolver el problema XOR. En este caso, las conexiones inhibitorias se sitúan en la primera capa para detectar los patrones 01 y 10.

Neurona	Entradas	Sesgo b
h_1	x_1	x_2
h_2	x_1	x_2
y_{xor}	h_1	$-2h_2$
		0

Stephen Grossberg, entonces en el MIT, propuso varios modelos de redes neuronales que podían modelar funciones no lineales.^{178,179}

El problema, no obstante, seguía siendo el mismo. ¿Cómo entrenamos este tipo de redes de forma automática, teniendo en cuenta que su topología es más compleja que la de un perceptrón simple? La solución, en el próximo capítulo...

Tabla 5: Una tercera solución al problema XOR, esta vez basada en el uso de unidades lineales rectificadas (ReLU).

¹⁷⁸ Stephen Grossberg. Pattern Learning by Functional-Differential Neural Networks with Arbitrary Path Weights. En Klaus Schmitt, editor, *Delay and Functional Differential Equations and their Applications*, pages 121–160. Academic Press, 1972. ISBN 978-0-12-627250-5. DOI: 10.1016/B978-0-12-627250-5.50009-5

¹⁷⁹ Stephen Grossberg. Contour Enhancement, Short Term Memory, and Constancies in Reverberating Neural Networks. *Studies in Applied Mathematics*, 52(3):213–257, 1973. ISSN 1467-9590. DOI: 10.1002/sapm1973523213

El algoritmo de propagación de errores

Las redes neuronales con una sola capa de parámetros ajustables están muy limitadas con respecto a lo que pueden hacer, como es el caso de los perceptrones. Por tanto, resulta natural ampliar la capacidad de una red neuronal añadiendo capas adicionales de neuronas. Desde el exterior de la red, sólo son visibles la primera y la última capa de una red multicapa: la capa de entrada y la capa de salida. Todas las demás capas son capas “ocultas”. Las capas adicionales se denominan, por tanto, capas ocultas porque no son visibles desde el exterior. No tiene mayor secreto el término.

En las capas ocultas de una red multicapa se suelen emplear neuronas con funciones de activación no lineales, si bien puntualmente utilizaremos capas lineales en aplicaciones particulares. Si todas las capas ocultas estuviesen formadas por capas lineales, no obstante, la capacidad de la red multicapa no sería superior a la de un humilde perceptrón: todas las capas lineales se podrían combinar en una única capa, también lineal. En definitiva, nuestra red multicapa siempre incluirá alguna capa oculta con unidades no lineales.

Para entrenar una red multicapa no nos sirve el algoritmo de aprendizaje del perceptrón, ya que éste sólo se puede emplear en redes de una única capa con parámetros ajustables. En la capa de salida, el error cometido por las neuronas de salida nos puede servir para ajustar sus pesos. En las capas ocultas, no obstante, no sabemos realmente cuál debe ser el valor de activación de las neuronas ocultas. Sin embargo, sí podemos observar cómo varía el error en función de cómo varían sus parámetros, lo que nos servirá para diseñar un algoritmo de entrenamiento de redes multicapa: el algoritmo de propagación de errores, backpropagation o, por abreviar, backprop.

Las redes neuronales artificiales organizadas por capas, sin realimentación, constituyen la topología de red neuronal más utilizada en la práctica. Su denominación oficial es *feed-forward neural networks* (FFNNs). En ocasiones, por razones históricas, también se denominan perceptrones multicapa (*MLPs: MultiLayer Perceptrons*), si bien esta denominación no es del todo correcta, ya que las redes multicapa no utilizan el algoritmo de aprendizaje del perceptrón. De hecho, tampoco suelen utilizar neuronas binarias o bipolares como los perceptrones de Rosenblatt, sino que suelen emplear neuronas sigmoidales y, a menudo en *deep learning*, unidades lineales rectificadas (*ReLUs: Rectified Linear Units*).

En una red multicapa de tipo *feed-forward*, las neuronas se organizan por capas. La arquitectura típica de una red neuronal multicapa consta, al menos, de tres capas:

- Capa de entrada.

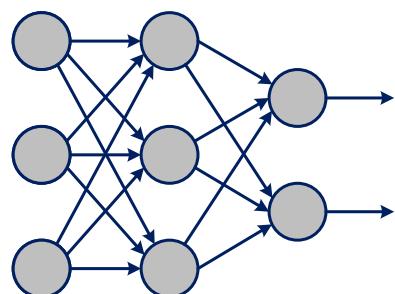


Figura 95: Red multicapa de tipo *feed-forward* con una única capa oculta.

- Capa oculta.
- Capa de salida.

Cada una de las capas recibe sus entradas de la capa inmediatamente anterior en la red (salvo la capa de entrada, que recibe sus entradas del exterior y se limita a propagarlas a la capa oculta). Desde el punto de vista complementario, las salidas de cada capa sirven de entradas a la capa inmediatamente posterior en la red multicapa.

Habitualmente, todas las salidas de una capa se distribuyen a todas las neuronas de la siguiente capa, formando capas completamente conectadas (*fully-connected layers*). No obstante, existen arquitecturas especializadas de redes neuronales artificiales en las que se utilizan otros patrones de conectividad. Por ejemplo, en las redes convolutivas que se emplean para procesar imágenes, se establecen patrones de conectividad local, por lo que una neurona no recibe entradas de todas las neuronas de la capa anterior, sino únicamente de aquéllas que forman parte de su campo receptivo local (su entorno más inmediato desde el punto de vista geométrico).

Cuando la red multicapa incluye más de una capa oculta, la red neuronal recibe el calificativo de profunda [*deep neural network*]. Éste es el origen del término *deep learning* (que podríamos traducir por “aprendizaje profundo” pero preferimos no hacerlo).

El uso de múltiples capas de neuronas ocultas es lo que permite a las redes neuronales artificiales utilizadas en *deep learning* extraer características más complejas a partir de otras características más simples. Las capas ocultas le permiten a la red neuronal construir un modelo interno de la forma en que los patrones de datos de entrada están relacionados con las salidas deseadas. Esta capacidad de aprendizaje de representaciones (*representation learning*) es lo que dota a las redes neuronales de un rendimiento sin par en aquellos dominios de aplicación en los que los datos de entrada se pueden interpretar construyendo una estructura jerárquica de características. Es el uso de múltiples capas ocultas donde reside el éxito del *deep learning* en problemas complejos como el reconocimiento de voz o la identificación de objetos en imágenes.

Obviamente, para que una red neuronal multicapa no se pueda reducir a una red de una única capa, que no es más que un clasificador lineal, las capas ocultas deben tener un comportamiento no lineal. Los niveles de actividad de las neuronas de cada capa vienen dados por una función no lineal de los niveles de actividad de las neuronas de la capa inferior. Esta no linealidad puede venir en forma de rectificador, como en los perceptrones de Rosenblatt; de función de activación sigmoidal, como la función logística o la tangente hiperbólica; o, incluso, de función gaussiana, como en las redes de funciones de base radial [*RBF: Radial Basis Function*]. Para algunas aplicaciones, como los problemas de clasificación, es habitual

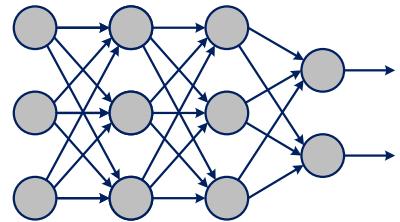


Figura 96: Red multicapa profunda de tipo *feed-forward*, con más de una capa oculta (dos en la figura).

utilizar modificaciones como la función *softmax*, una extensión de la función logística (en este caso, para la capa de salida de una red neuronal diseñada para resolver problemas de clasificación).

El entrenamiento de una red multicapa se suele realizar utilizando un algoritmo de propagación de errores hacia atrás denominado *backpropagation* (*backprop* para los amigos). El término *backpropagation* fue acuñado por Frank Rosenblatt como abreviatura de propagación de errores hacia atrás, si bien hoy suele hacer referencia a un algoritmo concreto de entrenamiento de redes neuronales multicapa que se popularizó tras la publicación en 1986 de un artículo de David Rumelhart, Geoffrey Hinton y Ronald Williams.¹⁸⁰ En realidad, la misma técnica había sido propuesta por Paul Werbos algo más de una década antes, en su tesis doctoral en Harvard.¹⁸¹

Técnicamente, el término *backpropagation* se refiere únicamente a la propagación de errores hacia atrás, no al algoritmo de entrenamiento completo. La propagación de errores hacia atrás permite calcular el gradiente del error con respecto a los diferentes parámetros de la red (esto es, cómo varía el error conforme varían los parámetros de la red). Combinado con una técnica de optimización como el gradiente descendente se obtiene un algoritmo de entrenamiento para redes multicapa. El gradiente obtenido por *backpropagation* indica en qué sentido han de modificarse los parámetros de la red y la técnica de optimización es la que realiza el ajuste de dichos parámetros. El gradiente se calcula para una función de error, también conocida como función de pérdida [*loss function*], y el método de optimización ajusta los pesos de la red con el objetivo de minimizar esa función de error o pérdida.

Aunque el método de aprendizaje consistente en combinar *backpropagation* con una técnica de optimización para minimizar una función de error dada es la forma más habitual de entrenar redes neuronales, no es la única que existe. Se pueden diseñar técnicas alternativas para ajustar los pesos de las conexiones sinápticas de una red neuronal, que son las que dotan de plasticidad a la red, ya sea biológica o artificial. La basada en *backpropagation* es un ejemplo típico de técnica de aprendizaje supervisado, si bien también existen técnicas no supervisadas y de aprendizaje por refuerzo con las que se pueden ajustar los parámetros de una red. ¿Cómo puede aprender una red neuronal sin supervisión externa? Esencialmente, realizando predicciones y comprobando, a posteriori, si esas predicciones resultaron ser correctas.

Los tres enfoques clásicos en los que se pueden englobar las distintas técnicas de aprendizaje propuestas son los siguientes:

- *Aprendizaje no supervisado* (correlación):

La red aprende a descubrir patrones en los datos de entrada sin supervisión, detectando correlaciones. Las correlaciones identificadas entre

¹⁸⁰ David E. Rumelhart, Geoffrey E. Hinton, y Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, October 1986a. doi: 10.1038/323533a0

¹⁸¹ Paul John Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974. URL <https://www.researchgate.net/publication/35657389>

Aunque el término *backpropagation* se asocie habitualmente a redes neuronales multicapa, en realidad describe una forma de calcular el gradiente que se puede utilizar en cualquier técnica de tipo numérico que requiera utilizar las derivadas parciales de una función con respecto a sus argumentos.

patrones de actividad neuronal y las entradas que los desencadenan se refuerzan de forma que, en el futuro, entradas similares desencadenarán patrones de actividad similares a los observados. Este tipo de aprendizaje resulta útil para extraer características de los datos de entrada y reconstruir los datos de entrada cuando éstos nos llegan con ruido.

- *Aprendizaje por refuerzo* (recompensas y castigos):

Ocasionalmente, la red recibe una recompensa o un castigo (p.ej. representada por la presencia de un neuromodulador como la dopamina). Esta recompensa o castigo indica si el comportamiento de la red fue el adecuado o no. El secreto de las técnicas de aprendizaje por refuerzo reside en su forma de resolver el problema de la asignación de crédito: dada una señal global, cómo modificar los pesos individuales de la red para mejorar su rendimiento en el futuro. Este tipo de aprendizaje se utiliza, por ejemplo, para que una red neuronal aprenda a jugar, ya sean videojuegos Atari o juegos de mesa como el Go, en los que la señal de recompensa o castigo sólo se recibe al final de la partida, en función de si ganamos o perdemos.

- *Aprendizaje supervisado* (gradiente descendente y *backpropagation*):

La red recibe una realimentación directa para cada uno de los ejemplos del conjunto de entrenamiento. Esa señal, en forma de salida deseada, se puede utilizar directamente para ajustar los parámetros de la red, tal como hacen las técnicas de optimización usadas con *backpropagation*.

Algunas de las técnicas de aprendizaje propuestas intentan ser fieles a lo que la neurociencia ha descubierto acerca del funcionamiento del cerebro, si bien otras son, simple y llanamente, el resultado del trabajo de ingenieros que intentan resolver problemas prácticos en el mundo real, sin pararse a pensar demasiado si su enfoque es biológicamente plausible. De hecho, el uso de *backpropagation* no parece demasiado plausible desde el punto de vista biológico, por lo que múltiples investigadores intentan descifrar cuál es el algoritmo de aprendizaje real que utilizan las redes neuronales biológicas. Para que resulte biológicamente plausible, se cree que el algoritmo de aprendizaje ha de basarse exclusivamente en la explotación de interacciones locales entre neuronas cercanas, en línea con el aprendizaje hebbiano, puede que con la ayuda de señales globales en forma de neuromoduladores como la dopamina. Biológicamente plausible o no, *backpropagation* ha resultado ser extremadamente útil en la práctica y sigue siendo la columna vertebral sobre la que se construyen todas las aplicaciones comerciales actuales del *deep learning*.

Entrenamiento de redes multicapa

Las redes multicapa de tipo *feed-forward* en ocasiones se denominan *backpropagation networks* porque el algoritmo de entrenamiento que se suele utilizar con ellas está basado en la propagación hacia atrás del error; esto es, en el uso de *backpropagation*.

Las redes multicapa usadas en *deep learning* tienen una capa de entrada, múltiples capas ocultas y una capa de salida. Desde un punto de vista formal, podemos verlas como una función matemática f que, a partir de un vector de entrada x , obtiene un vector de salida $y = f(x)$. Dado un conjunto de entrenamiento en forma de pares (x, y) , el objetivo del algoritmo de entrenamiento es ser capaz de aproximar la función f de forma que para cada posible entrada x se obtenga una salida $\hat{y} = f(x)$ lo más similar posible a la observada en el conjunto de entrenamiento.

Si los datos de entrada no incluyen ruido, nos bastaría con utilizar una red lo suficientemente grande como para que funcione como una simple tabla de consulta. Sin embargo, estaríamos sobreaprendiendo y la red neuronal no sería capaz de generalizar correctamente. Sería incapaz de proporcionar salidas apropiadas para entradas con las que no se hubiese encontrado en el conjunto de entrenamiento. Para que una red neuronal sea capaz de generalizar correctamente, tendremos que ajustar su capacidad hasta un nivel que resulte adecuado para la función que pretendemos modelar. Este ajuste de la capacidad se puede realizar, o bien ajustando los parámetros que determinan la topología de la red (habitualmente denominados hiperparámetros, para distinguirlos de los parámetros ajustados durante el entrenamiento de la red) o bien utilizando técnicas de regularización (cualquier técnica que nos permita reducir el error sobre el conjunto de prueba sin aumentar demasiado el error sobre el conjunto de entrenamiento). Pero no adelantemos acontecimientos todavía. El ajuste de los hiperparámetros de la red y el uso de técnicas de regularización los veremos en los dos próximos capítulos. En este nos centraremos en el algoritmo de entrenamiento de la red basado en *backpropagation*.

¿Cómo podemos entrenar una red multicapa? O, dicho de forma más precisa, ¿cómo ajustamos los pesos de las sinapsis correspondientes a las neuronas de las capas ocultas de una red multicapa? Necesitamos un algoritmo eficiente que nos permita adaptar todos los pesos de una red multicapa, no sólo los de la capa de salida.

El algoritmo que utilizamos para el perceptrón no puede extenderse para redes multicapa. Dicho algoritmo garantizaba su convergencia en un problema convexo (en el que las soluciones forman una región convexa del espacio, que denominamos hipercono de soluciones factibles al estudiar las propiedades del algoritmo de aprendizaje del perceptrón). Sin embargo, el aprendizaje de los pesos ocultos de una red multicapa es un problema

no convexo: la media de dos buenas soluciones puede no ser buena. Por tanto, el algoritmo de aprendizaje del perceptrón no nos vale. De ahí que insistamos en que es un error denominar perceptrón multicapa [MLP] a las redes multicapa de tipo *feed-forward*, por mucho que sea habitual en la práctica.

Aprender los pesos correspondientes a las neuronas de las capas ocultas equivale a aprender nuevas características. Estas características no están presentes en el conjunto de entrenamiento, por lo que resulta especialmente difícil su aprendizaje. ¿Por qué? Porque nadie nos dice directamente qué es lo que deberíamos aprender en cada una de las unidades ocultas que forman parte de las capas ocultas de nuestra red multicapa.

Así pues, dado que no sabemos cuál debería ser la salida “correcta” de las neuronas ocultas, intentaremos abordar el problema desde otro punto de vista. En vez de fijarnos en los cambios de los pesos de entrada que deberían producirse para obtener una salida deseada, nos fijaremos en los cambios que se producen en los niveles de activación de las neuronas ocultas si modificamos sus pesos. Dicho de otra forma, intentaremos determinar qué cambios en los niveles de activación de las neuronas ocultas son los que nos permiten acercarnos a las salidas deseadas en la capa de salida de la red. Este enfoque nos proporcionará una estrategia válida para resolver problemas no convexos.

Empecemos por un ejemplo de juguete: una sencilla neurona lineal.

Entrenamiento de neuronas lineales

Una neurona lineal, también conocida como filtro lineal, se limita a realizar el producto escalar de los vectores de pesos y entradas:

$$y = f(x) = \sum_i w_i x_i = w^\top x = w \cdot x$$

Si queremos ajustar los parámetros de una neurona lineal, su vector de pesos w , tendremos que definir una función de error, coste o pérdida que nos permita decidir qué conjuntos de pesos son mejores y cuáles son peores. Por ejemplo, podemos recurrir al error cuadrático medio [*MSE: Mean Squared Error*] medido sobre el conjunto de entrenamiento:

$$MSE = \frac{1}{n} \sum_j (t_j - y_j)^2$$

donde y_j es la salida proporcionada por la neurona lineal y t_j es el valor deseado, nuestro objetivo [*target*], tal como aparece en el conjunto de entrenamiento.

Dado que asumimos que el conjunto de entrenamiento es siempre el mismo (su tamaño no cambia durante el proceso de entrenamiento), minimizar el error cuadrático medio es equivalente a minimizar la suma

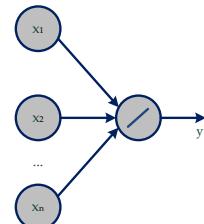


Figura 97: Una neurona lineal.

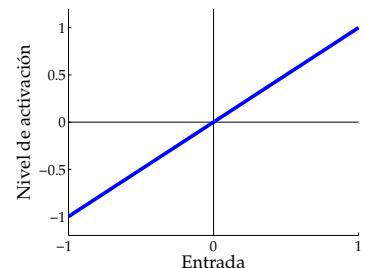


Figura 98: La función de activación de una neurona lineal.

de los errores al cuadrado [*SSE: Sum of Squared Errors*]:

$$SSE = \sum_j (t_j - y_j)^2$$

Dado que $y_j = w \cdot x_j$, el error depende directamente de los pesos de entrada de la neurona, que son los que tendremos que ajustar para reducir el error. Dado que hemos utilizado una función de error cuadrática, si dibujamos la función de error en función de un peso, obtendremos una parábola. Si lo hacemos en función de dos pesos, obtendremos un paraboloide de revolución (el resultado de rotar una parábola sobre su eje). En general, para n pesos, tendremos un paraboloide n-dimensional.

Con ayuda de una función de error, coste o pérdida, hemos reducido nuestro problema de aprendizaje supervisado inicial a un problema de optimización. Este problema de optimización consiste en determinar el valor más adecuado para los pesos de la neurona, aquél que reduce su error al mínimo de acuerdo a la función de coste predefinida.

Para encontrar el mínimo de la función de coste, tenemos dos opciones: resolver el problema de optimización desde el punto de vista analítico utilizando cálculo diferencial o resolverlo iterativamente utilizando cálculo numérico.

■ *Cálculo diferencial*

Si optamos por resolver analíticamente el problema de optimización, calculamos la derivada de la función de coste y la igualamos a cero. En el ejemplo anterior, para el que hemos supuesto una función de error cuadrática, su derivada será lineal y nos dará como resultado un sistema de ecuaciones lineales, con una ecuación para cada parámetro de la red.

Este sistema lo podremos resolver con facilidad siempre que trabajemos con redes pequeñas. No obstante, en las aplicaciones del *deep learning*, usualmente trabajaremos con redes neuronales con millones de parámetros, por lo que resolver un sistema de esas dimensiones puede resultar prohibitivo, aun siendo ecuaciones lineales. Cuando las funciones de activación no sean lineales, la resolución del problema no resultará tan sencilla, al dar lugar a sistemas de ecuaciones no lineales. Además, por distintos motivos, nos puede interesar utilizar otras funciones de error diferentes.

Aunque para el caso de las neuronas lineales podríamos resolver el problema analíticamente, la solución que obtendríamos sería difícil de generalizar. Dado que nuestro objetivo final es ser capaces optimizar una función de coste que involucrará dependencias no lineales complejas, el cálculo diferencial no siempre nos servirá. De hecho, la solución analítica a nuestro problema de optimización no siempre estará disponible. En consecuencia, optaremos por diseñar un algoritmo iterativo

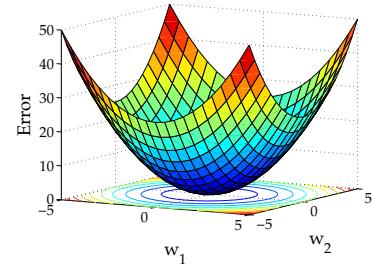


Figura 99: Superficie de error de una neurona lineal con dos entradas.

que, aunque menos eficiente, luego podamos generalizar a casos más complejos, con redes multicapa y funciones de activación no lineales.

- *Cálculo numérico*

Los métodos numéricos que nos permitirán resolver problemas de optimización complejos se basan en técnicas como el gradiente descendente, que podemos ver como la versión continua de las técnicas de ascensión de colinas que se utilizan en problemas discretos de optimización. Con la salvedad, obviamente, de que ahora estamos buscando el fondo del valle de la función de coste en lugar de la cima de la colina.

La ascensión de colinas es una técnica de búsqueda local, que hay quien compara con subir el Everest con una niebla espesa y amnesia. Con niebla, ya que somos incapaces de ver más allá de nuestro próximo paso. Con amnesia, porque no recordamos los lugares por los que hemos pasado ni volvemos atrás. El gradiente descendente es similar en ese sentido. En este caso, en vez de ascender estamos intentando descender, para llegar al fondo del valle que corresponde al mínimo de nuestra función de error, coste o pérdida. Hay una niebla extremadamente espesa, que hace que nuestra visibilidad sea mínima. Así que, dado que no podemos ver el camino que nos permite descender, hemos de tomar nuestras decisiones utilizando información local única y exclusivamente. La información local de la que disponemos es la pendiente del punto en el que nos encontramos, así que lo lógico sería seguir la dirección con mayor pendiente, hacia abajo. Esa dirección es precisamente la opuesta al gradiente de nuestra función de error, de ahí que el método se denomine gradiente descendente. Si estuviésemos buscando la cima de la montaña a ciegas, seguiríamos la dirección marcada por el gradiente, hablaríamos de maximización y estaríamos utilizando el gradiente ascendente.

A diferencia de lo que sucede cuando vamos de paseo por la montaña, evaluar la pendiente de la superficie donde nos movemos no es algo inmediato, sino que requiere realizar ciertos cálculos, que conllevan algo de tiempo. De forma que, en vez de evaluar instantáneamente el gradiente de la función y determinar en cada punto qué dirección seguir, iremos dando pequeños saltos en la dirección del gradiente descendente. El tamaño de esos saltos vendrá dado por la tasa de aprendizaje de nuestro algoritmo de aprendizaje, que deberá ser lo suficientemente grande para que no se eternice nuestra búsqueda del mínimo pero, a la vez, lo suficientemente pequeño para que no pasemos de largo ese mínimo. En otras palabras, la tasa de aprendizaje determina con qué frecuencia evaluamos la pendiente de la superficie de error en nuestra búsqueda de su mínimo.

Dado que estamos utilizando sólo información local, que lleguemos o no al mínimo global de nuestra función de coste dependerá de la

forma que tenga dicha función. Si la función es convexa, tendrá un único mínimo. Si damos pasos pequeños para ir acercándonos a ese mínimo, pero sin pasarnos, el gradiente descendente nos garantizará llegar al mínimo de la función. Si la función no es convexa, como suele ser habitual, el gradiente descendente podría en principio quedarse atascado en un mínimo local (o en una zona completamente plana de la función de coste, de la que tampoco sabríamos cómo salir al carecer de gradiente).

Se puede establecer una analogía entre el gradiente descendente y la gravedad. Imaginemos que, sobre la superficie de error, colocamos una bola. La bola tenderá a minimizar su energía potencial deslizándose sobre la superficie de error hasta llegar a su mínimo. Obviamente, asumimos que existe algo de fricción en el movimiento de la bola sobre la superficie. Esa fricción es la que permitirá que la bola se detenga en el mínimo de la función de error. En términos más formales, nuestro objetivo es que el movimiento de la bola hasta el mínimo sea un movimiento con amortiguamiento crítico, la forma más rápida de llegar al mínimo sin pasarnos, en el límite entre el sobreamortiguamiento [*overdamping*], que evita por completo oscilaciones, y las oscilaciones del amortiguamiento débil [*underdamping*], en las que la energía cinética de la bola le haría pasarse del mínimo y oscilar de forma amortiguada antes de detenerse en el mínimo. En cierta medida, el gradiente descendente no es más que simular ese escenario sobre la superficie de error, usando el gradiente para determinar la dirección del movimiento y la tasa de aprendizaje para determinar la velocidad con la que nos movemos siguiendo la dirección del gradiente descendente.

En la naturaleza, se encuentra esta forma de resolver problemas incluso en el comportamiento de organismos simples como bacterias y gusanos. Un gusano *C. elegans* dispone de una red neuronal que le permite recordar a qué temperatura se alimentó bien por última vez, temperatura que luego busca utilizando los sensores térmicos concentrados en su “cabeza”. En su movimiento, la trayectoria del gusano sigue un camino aleatorio guiado por los gradientes de temperatura y de concentración de bacterias de las que se alimenta, como *E. coli*. La bacteria *E. coli*, una balsa de apenas $1 \times 3\mu\text{m}$ al lado del superpetrolero *C. elegans*, también sigue en su movimiento un gradiente, el de concentración de los nutrientes de los que se alimenta, como glucosa o lactosa. La misma estrategia que emplea un paramecio, un protozoo unicelular de $350 \times 50\mu\text{m}$, miles de veces más grande que *E. coli* y que se mueve 50 veces más rápido que esta última: un auténtico tiburón en un mar de bacterias. Para lograr su velocidad de respuesta, la membrana del paramecio utiliza señales eléctricas para indicar a sus cilios, en milisegundos, cambios de dirección al chocar con un ob-

Las máquinas de aprendizaje extremo [ELMs: *Extreme Learning Machines*] son un tipo particular de redes neuronales con una sola capa de pesos ajustables, lo que garantiza que el problema de optimización es convexo y se puede resolver de forma exacta con técnicas analíticas, por lo que su entrenamiento es mucho más eficiente que el de las redes multicapa entrenadas con *backpropagation* y gradiente descendente.

Guang-Bin Huang, Qin-Yu Zhu, y Chee-Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1):489 – 501, 2006. ISSN 0925-2312. DOI: 10.1016/j.neucom.2005.12.126

Caenorhabditis elegans es un nematodo de apenas un milímetro de longitud. Es el organismo más utilizado en estudios bioquímicos y genéticos por ser transparente (lo que facilita observarlo al microscopio), hermafrodita (simplifica la obtención de individuos con mutaciones recesivas) y un breve período de vida, de 2 a 3 semanas, lo que permite realizar experimentos y obtener resultados con rapidez. Con 959 células, fue el primer organismo multicelular cuyo genoma se secuenció por completo, con 6 pares de cromosomas, 97 millones de pares de nucleótidos y más de 19000 genes, el 40 % de los cuales compartimos con ellos. Su red neuronal, formada por 302 neuronas, también es la más estudiada. Tras décadas de esfuerzos se dispone de un mapa de su conectoma preciso al 93 %. El 7 % de error da una idea de las dificultades técnicas que entraña el análisis de las conexiones neuronales de un organismo biológico.

Peter Sterling y Simon Laughlin. *Principles of Neural Design*. MIT Press, 2015. ISBN 0262028700

táculo, ya que el mecanismo de difusión química empleado por *E. coli* sería demasiado lento para una bacteria del tamaño del paramecio. Sin duda, un precursor de las redes neuronales de los sistemas nerviosos que encontramos en los organismos pluricelulares, desde el *C. elegans* hasta el ser humano.

En resumen, aunque el problema del ajuste de los pesos de una neurona lineal lo podríamos resolver analíticamente, la solución analítica será difícil de generalizar, por lo que diseñaremos un algoritmo iterativo, menos eficiente pero más fácil de generalizar. ¿Por qué no recurrimos a técnicas analíticas? Desde el punto de vista de un científico, que construye para estudiar, porque aspiramos a descubrir un método que las neuronas biológicas puedan utilizar (y resulta difícil imaginarlas resolviendo analíticamente un problema de optimización). Desde el punto de vista de un ingeniero, que estudia para construir, porque nuestro objetivo final es generalizar la técnica para que sea aplicable a redes multicapa con unidades no lineales (algo inviable para redes complejas desde un punto de vista meramente analítico).

Un ejemplo intuitivo

Imaginemos que nos suscribimos a un servicio de vídeo bajo demanda [VoD: Video on Demand], como los ofrecidos por Netflix, Amazon Video, Apple TV o Hulu. Como somos propensos a darnos atracones de nuestras series favoritas, en sesiones maratonianas de *binge-watching*, de vez en cuando la factura de nuestro servicio nos da alguna que otra sorpresa.

Hipotéticamente, supongamos que nuestro proveedor de vídeo utiliza una tarifa según la cual la visualización de una película de estreno, una película clásica o un episodio de una serie de televisión supone un cargo adicional en nuestra factura mensual, diferente en función del tipo de producto consumido. Además, en nuestro experimento, asumiremos que hemos sufrido un ataque de amnesia selectiva que nos impide acordarnos de que las tarifas se pueden consultar fácilmente en la web de nuestro proveedor. Esa amnesia selectiva, no obstante, nos ha traído a la memoria lo que aprendimos de matemáticas hace años, cuando estábamos en el colegio.

Cada mes, nuestra compañía de VoD carga en nuestra cuenta bancaria el importe asociado a nuestro consumo de contenidos digitales en el mes anterior. Somos perfectamente conscientes de las películas y series que vemos casi a diario. También conocemos el importe total que se nos ha facturado a final de mes, ya que nos aparece en el extracto del banco. Sin embargo, nuestra amnesia nos impide recordar dónde consultar las tarifas vigentes. ¿Cómo podemos determinar el importe individual asociado a cada una de nuestras experiencias audiovisuales, sin tratar nuestro extraño caso de amnesia selectiva?

Los proveedores de vídeo bajo demanda no utilizan realmente este modelo de facturación, sino que, por motivos comerciales, suelen ofrecer distintos tipos de tarifas planas. No obstante, supongamos que facturan por consumo real en nuestro hipotético ejemplo...

Ya dijimos que habíamos sufrido un peculiar ataque de amnesia selectiva, aunque en este caso puede ayudarnos el hecho de que nunca se nos olvide evaluar lo que vemos en IMDB [*Internet Movie Database*], <http://www.imdb.com>.

Sabemos que nuestro consumo mensual de productos audiovisuales consta de películas de estreno, cine clásico y series de televisión. El importe de nuestro consumo aparece cargado en nuestra cuenta, como resultado de aplicar una sencilla relación lineal definida sobre los precios y cantidades consumidas de cada tipo de servicio:

$$\text{total} = w_{\text{estreno}} \times x_{\text{estreno}} + w_{\text{clásico}} \times x_{\text{clásico}} + w_{\text{TV}} \times x_{\text{TV}}$$

donde w_i corresponde al precio individual asociado al tipo de servicio i y x_i es el número de unidades consumidas del tipo de servicio i . Esto es, los precios que queremos descubrir son los pesos y las cantidades conocidas son las entradas de nuestra neurona lineal.

Todos los meses consumimos esos tres tipos de servicios y, para descubrir su precio individual, emplearemos una estrategia iterativa. La idea es que, analizando las facturas correspondientes a varios meses consecutivos (en los que asumimos que los precios se mantienen estables), seamos capaces de determinar los precios individuales de películas de estreno, cine clásico y episodios de series de televisión.

Inicialmente, partimos de una estimación, más o menos aleatoria, de los precios asociados a cada consumo. Decimos más o menos aleatoria porque, siguiendo un enfoque bayesiano, nuestra estimación tiene en cuenta información que reside en nuestro subconsciente que nos permite afinar nuestra estimación a priori de los precios. En nuestro caso, supongamos que esa estimación inicial es:

$$\begin{aligned} w_{\text{estreno}} &= 3 \\ w_{\text{clásico}} &= 3 \\ w_{\text{TV}} &= 2 \end{aligned}$$

Asumimos, como suelen hacer los economistas, que los precios vienen dados en unidades monetarias (u.m.). Mentalmente, puede interpretar esas cantidades como si apareciesen en euros (España), dólares (Ecuador, El Salvador, Puerto Rico), pesos (Chile, Colombia, Argentina, Uruguay, Cuba, República Dominicana), reales (Brasil), colones (Costa Rica), quetzales (Guatemala), soles (Perú), bolívares (Venezuela) o cualquier otra moneda que sea de su interés (igual prefiere realizar este ejercicio usando dracmas, minas, talentos, denarios, sextercios, peniques, chelines, libras o reales de a ocho, quién sabe). Obviamente, para conseguir un ejercicio realista, deberá multiplicar el valor dado en unidades monetarias genéricas por un tipo de cambio adecuado para que los precios resultantes sean plausibles.

Aunque no lo sabemos, el precio real de los servicios que tenemos contratados es el siguiente:

$$\begin{aligned} w_{\text{estreno}}^* &= 4 \\ w_{\text{clásico}}^* &= 2 \\ w_{\text{TV}}^* &= 1 \end{aligned}$$

Así que, si durante un mes vemos 4 películas de estreno los viernes, 6 películas de cine clásico y 12 episodios de series de televisión en dos sesiones maratonianas, la factura que nos llega a final de mes es del siguiente importe:

$$\begin{aligned}factura &= \sum x_i w_i^* \\&= 4 \times 4 + 6 \times 2 + 12 \times 1 \\&= 16 + 12 + 12 \\&= 40 \text{ u.m.}\end{aligned}$$

Dada nuestra estimación inicial de precios, esperábamos recibir una factura de un importe diferente:

$$\begin{aligned}estimación &= \sum x_i w_i \\&= 4 \times 3 + 6 \times 3 + 12 \times 2 \\&= 12 + 18 + 24 \\&= 54 \text{ u.m.}\end{aligned}$$

Esto es, hemos cometido un error de estimación de +14 u.m.:

$$error = estimación - factura = +14 \text{ u.m.}$$

A continuación, nos proponemos ajustar los precios estimados para que cuadren con los importes facturados. Para ello, utilizaremos una regla de actualización de los pesos conocida por el nombre de regla delta. La regla modifica los pesos en dirección opuesta al error cometido, con el objetivo de reducir dicho error. Formalmente, aplicamos la siguiente fórmula de corrección de los pesos:

$$\Delta w_i = -\eta \text{ error } x_i = \eta(factura - estimación)x_i$$

donde η [eta] es la tasa de aprendizaje utilizada para controlar el tamaño de los ajustes realizados en cada iteración.

Para facilitar los resultados numéricos en nuestro ejemplo, donde el error de nuestra estimación era de 14 u.m., supongamos que utilizamos una tasa de aprendizaje $\eta = 1/140$:

$$\Delta w = (-0.4, -0.6, -1.2)$$

Observe que en la regla delta, se multiplica el error de la neurona por una constante, la tasa de aprendizaje, que hemos escogido casualmente para que $\eta \text{ error} = 0.1$. Esto es, la corrección del vector de pesos es proporcional al error y paralela al vector de entradas $(4, 6, 12)$, con una constante de proporcionalidad que viene dada por la tasa de aprendizaje. Tras modificar los pesos, nuestra estimación de los precios pasará a ser:

$$w(1) = w(0) + \Delta w = (2.6, 2.4, 0.8)$$

Normalmente, las señales de error se representan como una señal de realimentación que recibe el sistema cuyo comportamiento deseamos controlar: la diferencia entre la salida deseada y la salida obtenida. En nuestro ejemplo, esa señal de error sería de -14 u.m. y se usaría directamente para corregir el comportamiento del sistema. No obstante, aquí usamos una interpretación más intuitiva del error, a la que debemos añadir un signo menos en la corrección de los pesos. En realidad, cuando se utiliza una función de error cuadrática, como aquí hacemos de forma implícita, da igual en qué sentido calculemos la diferencia, ya que el resultado será el mismo en ambos casos.

Si realizamos una nueva estimación del importe de la factura utilizando los precios actualizados, vemos que el error se ha reducido:

$$\begin{aligned} \text{estimación} &= \sum x_i w_i \\ &= 4 \times 2.6 + 6 \times 2.4 + 12 \times 0.8 \\ &= 10.4 + 14.4 + 9.6 \\ &= 34.4 \text{ u.m.} \end{aligned}$$

El ajuste de los pesos ha sido algo drástico, ya que ahora cometemos un error de -5.6 u.m. en sentido opuesto al error inicial (+14 u.m.). En términos absolutos, nuestra modificación de los precios ha permitido mejorar la estimación del importe de la factura, si bien nuestra estimación del precio de las películas de estreno no sólo no ha mejorado, sino que ha empeorado: su precio real es 4 u.m., antes creímos que costaban 3 u.m. y ahora pensamos que cuestan 2.6 u.m. (una estimación peor que la inicial porque hemos modificado todos los precios en el mismo sentido aunque el error cometido era diferente para cada uno de ellos). El error total, no obstante, se reduce porque hemos ajustado mejor el precio de las películas clásicas (2.4 estimado frente a 2.0 real) y el de los episodios de series de televisión (0.8 estimado frente a 1.0 real).

Si repetimos el proceso de ajuste de pesos varias veces usando facturas de meses posteriores podríamos intentar afinar más, con el objetivo de descubrir los precios reales de los servicios que tenemos contratados. Por ejemplo, si el siguiente mes realizásemos el mismo consumo, el error de nuestra estimación sería de -5.6 u.m., como acabamos de ver. El ajuste de los pesos se realizaría de acuerdo a la expresión:

$$\Delta w_i = -\eta \text{error } x_i = -(1/140)(-5.6)x_i = +0.04x_i$$

por lo que

$$\Delta w = (0.16, 0.24, 0.48)$$

y los pesos en la siguiente iteración tomarían los valores

$$w(2) = w(1) + \Delta w = (2.6, 2.4, 0.8) + (0.16, 0.24, 0.48) = (2.76, 2.68, 1.28)$$

Nuestra estimación pasaría a ser

$$\begin{aligned} \text{estimación} &= \sum x_i w_i \\ &= 4 \times 2.76 + 6 \times 2.68 + 12 \times 1.28 \\ &= 11.04 + 16.08 + 15.36 \\ &= 42.48 \text{ u.m.} \end{aligned}$$

El error se reduce a sólo 2.48 u.m., mucho menor que el estimado inicialmente tras sólo dos iteraciones de nuestro algoritmo.

Para ser capaces de ajustar correctamente los pesos, tendríamos que repetir el proceso de forma iterativa, para diferentes vectores de entrada. Usar siempre el mismo vector de entrada no nos permitiría discriminar correctamente el error cometido por los diferentes parámetros de nuestra neurona lineal, por la correlación existente entre los vectores de entrada. Esa correlación debería no estar presente en nuestros datos de entrenamiento si queremos corregir la subestimación sistemática que estamos realizando del precio de las películas de estreno (cuyo precio real, recordemos, era de 4 u.m.).

La regla delta

En el ejemplo anterior, usamos una función de corrección de los parámetros de la neurona lineal que, en notación vectorial, podemos representar de la siguiente forma:

$$\Delta w = -\eta \text{ error } x_j = \eta(t_j - y_j)x_j$$

donde x_j es un vector de entrada, t_j corresponde a la salida deseada para la entrada x_j (nuestro objetivo), y_j es la salida proporcionada por la neurona al recibir la entrada x_j (una estimación de su valor real, t_j) y, por último, Δw es la modificación que debemos aplicar a los pesos de la neurona de acuerdo con una constante η a la que llamamos tasa de aprendizaje.

¿De dónde proviene esta regla de actualización de los pesos? Para descubrirlo, partamos de una función de error que pretendemos minimizar. Por ejemplo, podríamos utilizar el error cuadrático medio, que realiza una media aritmética de los residuos al cuadrado:

$$E_{MSE} = \frac{1}{2} \frac{1}{n} \sum_j (t_j - y_j)^2$$

Por convención, hemos añadido un factor adicional, $1/2$, que utilizamos para simplificar los cálculos cuando se derive la función de error. En realidad, podemos prescindir también del factor $1/n$ y quedarnos directamente con la suma de los residuos al cuadrado:

$$ESSE = \frac{1}{2} \sum_j (t_j - y_j)^2$$

Ambas medidas de error las calculamos sobre los n ejemplos del conjunto de entrenamiento, si bien también podemos realizar una estimación puntual del error sobre un caso concreto j del conjunto de datos de entrenamiento. En ese caso, la estimación puntual del error tiene la forma

$$E_j = \frac{1}{2} (t_j - y_j)^2$$

por lo que nuestra medida de error sobre el conjunto de entrenamiento la podemos reescribir como

$$E = ESSE = \sum_j E_j$$

¿Por qué hemos utilizado un error cuadrático? Después de todo, si normalmente estamos interesados en resolver problemas de aprendizaje supervisado, ¿no tendría sentido utilizar como función de error el número de ejemplos correctamente clasificados? ¿Por qué no intentar maximizar directamente esa cantidad en vez de minimizar una medida indirecta como el error cuadrático? El problema es que el número de ejemplos

El factor adicional $1/2$ aparece si establecemos la correspondencia entre minimización de la función de error y la maximización del estimador MAP [*maximum a posteriori*] que se deriva del teorema de Bayes cuando asumimos que el error cometido en la estimación sigue una distribución gaussiana.

Simon Haykin. *Neural Networks and Learning Machines*. Pearson, 3rd edition, 2008. ISBN 0131471392

clasificados no nos proporciona una función continua y derivable que podamos definir sobre los pesos y las entradas de la red. Nuestro objetivo es conseguir una función que podamos evaluar cuando introduzcamos pequeños cambios en los parámetros de la red, de forma que los cambios observados en la función de evaluación puedan utilizarse para determinar el sentido en el que se han de corregir esos parámetros. Si utilizásemos el número de ejemplos clasificados correctamente, la mayor parte de los cambios que realizásemos sobre los pesos de la red no inducirían cambio alguno en la clasificación, por lo que no obtendríamos una señal útil para determinar cómo corregir esos pesos de forma que mejore el rendimiento de la red. Al usar el error cuadrático como señal de error, obtenemos una función continua y derivable que nos indica cómo ir modificando los pesos. Veamos cómo...

Para minimizar el error, podemos calcular su derivada (gradiente en el caso vectorial) y realizar una corrección de los pesos en el sentido opuesto al indicado por esa derivada.

En el caso unidimensional, la derivada del error será positiva cuando nos encontremos en el lado derecho de la parábola definida por la función de error cuadrático. Por tanto, como deseamos minimizar ese error, deberemos disminuir el valor actual del peso w , con el objetivo de acercarnos al valor óptimo w^* . El movimiento lo hacemos, pues, en sentido contrario al indicado por la derivada del error con respecto al peso.

En el caso multidimensional, nuestra superficie de error tendrá forma de paraboloide multidimensional y el gradiente desempeñará el papel de la derivada en el caso unidimensional.

Dado que estamos utilizando neuronas lineales, su salida y es una combinación lineal de los pesos w y sus entradas x :

$$y = w^\top x = w \cdot x = \sum_i w_i x_i$$

Las derivadas parciales del error con respecto a cada uno de los pesos de la neurona vienen dadas por la expresión

$$\frac{\partial E}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial y} \frac{\partial y}{\partial w_i}$$

donde hemos utilizado la definición de nuestra función de error como suma de errores instantáneos y la regla de la cadena para calcular la derivada del error con respecto a los pesos como un producto de dos derivadas: la del error con respecto a la salida y la de la salida con respecto a los pesos.

La derivada de la estimación instantánea del error con respecto a la salida de la neurona es

$$\frac{\partial E_j}{\partial y} = -(t_j - y_j)$$

En realidad, cuando pretendamos resolver problemas de clasificación con redes neuronales, utilizaremos otra función de error, la entropía cruzada, que analizaremos detalladamente cuando estudiemos las capas *softmax*, que sustituirán a las capas de salida de nuestra red neuronal cuando pretendamos resolver problemas de clasificación. Por ahora, no obstante, nos quedaremos con el error cuadrático para entender bien los fundamentos de los algoritmos de entrenamiento de redes neuronales multicapa.

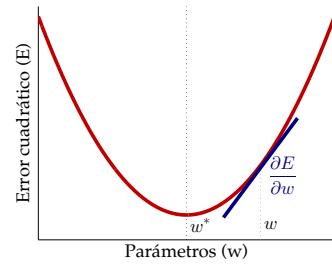


Figura 100: El error cuadrático en función de un parámetro (peso) de la neurona lineal. El mínimo del error se consigue para la asignación óptima del peso (w^*). La derivada del error para el valor actual del peso (w) nos indica en qué sentido debemos corregir ese valor: el opuesto al indicado por la derivada del error en ese punto.

La derivada de la salida de la neurona con respecto al peso w_i es

$$\frac{\partial y}{\partial w_i} = x_i$$

Sustituyendo ambas derivadas en la expresión de la derivada del error obtenemos:

$$\frac{\partial E}{\partial w_i} = - \sum_j (t_j - y_j) x_{ji}$$

donde x_{ji} representa el valor de la entrada i para el patrón de entrenamiento j .

Para simplificar las expresiones y prescindir del uso excesivo de subíndices, utilizaremos una notación vectorial. En primer lugar, definimos el operador gradiente ∇ (o ∇_w si queremos ser explícitos a la hora de indicar con respecto a qué variables estamos derivando):

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \dots, \frac{\partial}{\partial w_n} \right]$$

Entonces, el vector gradiente de nuestra función de error con respecto a los pesos, ∇E o $\nabla_w E$, viene dado por:

$$\nabla E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_n} \right]$$

Las derivadas parciales del error de nuestra neurona lineal, en notación vectorial, las podemos representar como un gradiente:

$$\nabla E = - \sum_j (t_j - y_j) x_j$$

Ese gradiente es el que utilizamos para ajustar los parámetros de la red con el objetivo de minimizar su error, moviéndonos en sentido opuesto al indicado por el vector del gradiente:

$$\Delta w = -\eta \nabla E = \eta \sum_j (t_j - y_j) x_j$$

Observe que, si en vez de utilizar el error sobre todo el conjunto de entrenamiento, nos quedamos únicamente con la estimación instantánea del error para un patrón de entrenamiento dado, tenemos

$$\nabla E_j = -(t_j - y_j) x_j$$

Sustituyendo en la expresión de actualización de los pesos,

$$\Delta w = -\eta \nabla E_j = \eta (t_j - y_j) x_j$$

que es exactamente la misma expresión que utilizamos para corregir los pesos de la neurona lineal en el ejemplo de la sección anterior, en el que utilizamos una estrategia de entrenamiento que se conoce con el nombre de aprendizaje *online*.

La alternativa es utilizar aprendizaje por lotes [*batch learning*], en el que sólo se actualizan los pesos una vez que hemos visto todos los ejemplos del conjunto de entrenamiento y calculado el error E para el conjunto de entrenamiento completo.

Resumiendo, en cada iteración del algoritmo, ajustamos los pesos en proporción al gradiente del error cometido:

- En notación vectorial:

$$\Delta w = -\eta \nabla E$$

- Individualmente, para cada peso:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

El algoritmo del gradiente descendente es un algoritmo iterativo, que repetidamente calcula el gradiente del error ∇E y realiza ajustes en los parámetros de la red en sentido opuesto a dicho gradiente, lo que nos hace caer por la pendiente de la superficie de error en dirección al mínimo.

¿Por qué es correcto este ajuste de los parámetros de la red cuando queremos minimizar el error? Porque podemos aproximar la variación que se produce en el error, ΔE , para pequeños cambios de los parámetros, Δw :

$$\Delta E \approx \frac{\partial E}{\partial w_1} \Delta w_1 + \frac{\partial E}{\partial w_2} \Delta w_2 + \dots + \frac{\partial E}{\partial w_n} \Delta w_n$$

o, más sucintamente, utilizando el gradiente y un producto escalar:

$$\Delta E \approx \nabla E \cdot \Delta w$$

Cuando realizamos el ajuste de los pesos utilizando la expresión $\Delta w = -\eta \nabla E$, de acuerdo a nuestra aproximación de la variación del error, el error variará de la siguiente forma:

$$\Delta E \approx \nabla E \cdot \Delta w = \nabla E \cdot (-\eta \nabla E) = -\eta \nabla E \cdot \nabla E = -\eta \|\nabla E\|^2$$

Como la tasa de aprendizaje η es una constante positiva ($\eta > 0$) y la norma euclídea del gradiente del error es también mayor o igual que 0 ($\|\nabla E\|^2 \geq 0$), la variación del error ΔE será menor o igual que 0 ($\Delta E \leq 0$). Esto es, el error siempre disminuye dentro de la aproximación que hemos realizado, para la que necesitamos que la tasa de aprendizaje η sea lo suficientemente pequeña.

Otra forma equivalente de intepretar por qué funciona el gradiente descendente es realizar una expansión en serie de Taylor de primer orden de la función de error en torno a $w(t)$ para aproximar $w(t+1) = w(t) + \Delta w(t)$:

$$E(w(t+1)) \approx E(w(t)) + \nabla E(w(t)) \cdot \Delta w(t)$$

Para un valor pequeño de η , es justificable el uso que hemos hecho de $\Delta w(t) = -\eta \nabla E(w(t))$, expresión de la que se obtiene

$$\begin{aligned} E(w(t+1)) &\approx E(w(t)) - \eta \nabla E(w(t)) \cdot \nabla E(w(t)) \\ &= E(w(t)) - \eta \|\nabla E(w(t))\|^2 \end{aligned}$$

Por el mismo argumento que antes, se observa que, para tasas de aprendizaje positivas lo suficientemente pequeñas, la función de error decrece cuando pasamos de una iteración a la siguiente.

El método del gradiente descendente converge a la solución óptima w^* lentamente. De hecho, el valor concreto que utilicemos para la tasa de aprendizaje η tiene consecuencias con respecto a la velocidad de convergencia (y a la propia convergencia del algoritmo):

- Cuando la tasa η es pequeña, los pesos siguen una trayectoria sobre-amortiguada [*overdamped*], en la que lentamente nos vamos acercando al óptimo de la función de error.
- Cuando la tasa η es grande, los pesos siguen un camino en zig-zag, con oscilaciones propias de un oscilador armónico con amortiguamiento débil [*underdamping*].
- Cuando la tasa η excede de un valor crítico, las oscilaciones, en vez de irse amortiguando, van ampliando su magnitud de una iteración a la siguiente, por lo que el algoritmo pasa a ser inestable y diverge (no converge a la solución óptima).

Para conseguir que el método del gradiente descendente funcione correctamente, tendremos que elegir una tasa de aprendizaje lo suficientemente pequeña, que nos permita ir acercándonos a la solución óptima. No obstante, también nos interesa que la convergencia al óptimo se realice lo más rápidamente posible, lo que implica aumentar el valor de la tasa de aprendizaje. La elección de un valor adecuado para la tasa de aprendizaje es, pues, un factor crítico a la hora de aplicar con éxito el método del gradiente descendente para entrenar una red neuronal. Lo suficientemente pequeña para que el algoritmo converja y lo suficientemente grande para que lo haga rápidamente, sin llegar a divergir.

Si volvemos una vez más a la representación de la superficie de error en forma de paraboloides (o paraboloides n-dimensional en el caso general), podemos realizar secciones transversales de dicha superficie de error y representarlas en un mapa de contorno (como el mapa de las isobáras del pronóstico del tiempo).

Dibujamos secciones transversales horizontales de la función de error para una neurona lineal con dos entradas, para que la gráfica resultante sea bidimensional. Esas secciones transversales horizontales del paraboloides serán elípticas, mientras que las secciones verticales serían parábolas.

Hay un sentido en el que el gradiente descendente es la estrategia óptima para buscar el mínimo de la función de error. Supongamos que queremos realizar un cambio en los parámetros Δw que disminuya el error E lo máximo posible. Este cambio es equivalente a minimizar $\Delta E \approx \nabla E \cdot \Delta w$. Si restringimos el tamaño de la modificación $\|\Delta w\| = \epsilon$ para un $\epsilon > 0$, obligamos a dar un pequeño paso, de longitud fija, que disminuya el error E tanto como sea posible. Se puede demostrar, utilizando la desigualdad de Cauchy-Schwarz, que ya utilizamos para demostrar la convergencia del perceptor, que la selección de Δw que minimiza $\nabla E \cdot \Delta w$ es $\Delta w = -\eta \nabla E$, donde $\eta = \epsilon / \|\nabla E\|$. En definitiva, el gradiente descendente se puede ver como la forma de ir dando pequeños pasos en la dirección que disminuye más el error E de forma inmediata. El problema es que, en la práctica, para reducir el tiempo de ejecución del algoritmo, nos gustaría no tener que ir dando pasos infinitesimalmente pequeños.

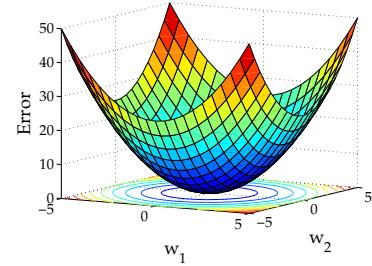


Figura 101: Superficie de error de una neurona lineal. La representación gráfica utiliza únicamente dos dimensiones (dos pesos de entrada) para visualizar el error en tres dimensiones.

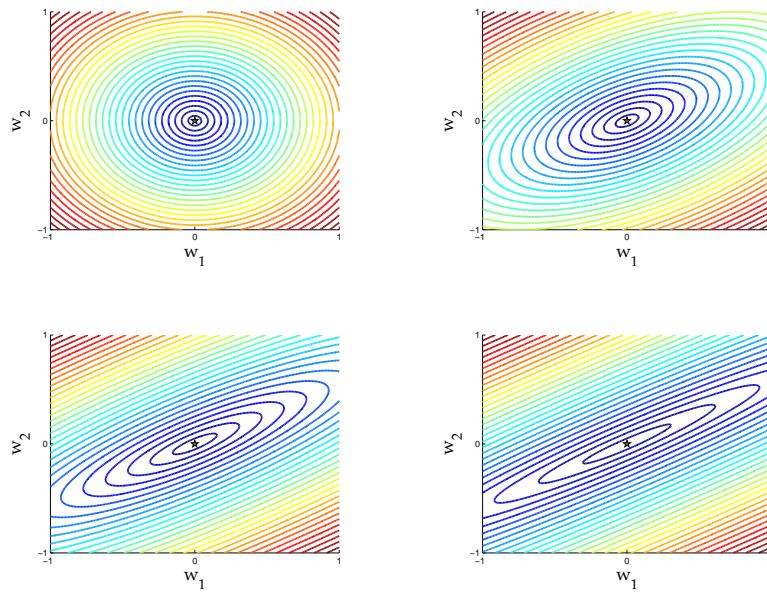


Figura 102: Secciones transversales de la superficie de error conforme aumenta la correlación entre variables. Cuanto mayor sea ésta, más alargadas serán las elipses y menos útil resultará el gradiente (perpendicular a ellas) para guiarnos en nuestro camino hacia el mínimo.

En redes multicapa con elementos no lineales la superficie de error puede ser mucho más complicada. Aun así, localmente, una aproximación cuadrática de la superficie de error puede servirnos en cualquier caso para entender mejor el funcionamiento del gradiente descendente.

La dirección del gradiente será perpendicular a las líneas de contorno asociadas a las secciones transversales horizontales de la superficie de error. Dado que esas secciones transversales horizontales son elípticas y el gradiente es perpendicular a las elipses en cada punto, el gradiente no apuntará necesariamente hacia el mínimo de la función de error. De hecho, cuando la elipse es muy alargada, la dirección puede llegar a ser prácticamente perpendicular a la dirección hacia el mínimo de la función de error: el vector del gradiente se descompondrá en un componente muy grande paralelo al eje menor de la elipse y un componente muy pequeño paralelo al eje mayor de la misma.

¿Por qué puede ser muy lento el aprendizaje al utilizar el gradiente descendente? Si existe correlación entre las variables de entrada, las secciones transversales de la superficie de error serán elípticas, con unas elipses más alargadas cuanto mayor sea la correlación entre las variables. En términos del ejemplo de la sección anterior, si lo nuestro son las sesiones dobles de cine en las que combinamos una película de estreno con una película clásica, la correlación existente entre las dos variables de entrada hará difícil que sepamos repartir el importe de la factura entre películas de un tipo y de otro. Simplemente, no sabremos cómo repartir el error cometido entre los pesos asociados a ellas (un problema de asignación de crédito). Como consecuencia, la convergencia del algoritmo de aprendizaje hacia la solución que minimiza el error en nuestra estimación puede llegar

a ser extremadamente lenta.

En el análisis anterior, hemos asumido que realizamos una estimación del gradiente utilizando todo el conjunto de entrenamiento, empleando aprendizaje por lotes [*batch learning*]. Cuando utilizamos aprendizaje *online*, se realiza una estimación instantánea del error para un único ejemplo del conjunto de entrenamiento y se utiliza esta estimación instantánea para estimar el gradiente y ajustar los parámetros de la red. El gradiente así obtenido dependerá del ejemplo concreto que se haya seleccionado. Generalmente, los datos del conjunto de entrenamiento se van seleccionando al azar, para lo que basta con barajarlos antes de cada recorrido del conjunto de entrenamiento. Esta selección al azar del ejemplo con el que se ajustan los pesos es un proceso estocástico, de ahí que se denomine gradiente descendente estocástico cuando usamos aprendizaje *online*.

- Cuando utilizamos aprendizaje por lotes y un valor adecuado para la tasa de aprendizaje, el movimiento hacia el mínimo se realizará de forma progresiva, siguiendo una trayectoria amortiguada. La regla delta por lotes cambia los pesos en proporción a las derivadas del error sumadas para todos los ejemplos del conjunto de entrenamiento. Como la estimación del gradiente será buena, el aprendizaje por lotes sigue la trayectoria asociada al gradiente real de la función de error, perpendicular a las secciones transversales de la superficie de error.
- Cuando utilizamos aprendizaje *online*, la estimación del gradiente fluctuará en función del ejemplo particular que se haya escogido en cada momento. La trayectoria del vector de pesos será zigzagueante, más o menos en línea con el gradiente descendente, pero no siempre en su dirección. En la versión *online* de la regla delta, que es la que utilizamos en el ejemplo de la sección anterior, modificamos el vector de pesos multiplicando la tasa de aprendizaje por el error residual y el vector de entrada asociado al ejemplo. La dirección de la actualización será menos fiable que la seguida cuando se emplea aprendizaje por lotes pero, como realizamos una actualización de los pesos cada vez que le presentamos un ejemplo a la red, la convergencia del algoritmo suele ser más rápida en términos de tiempo de CPU.

Entrenamiento de neuronas sigmoidales

Hemos visto cómo ajustar los parámetros de una neurona lineal de forma que se minimice el error cometido por ella utilizando la técnica del gradiente descendente. Nuestro siguiente paso es ver cómo podemos utilizar la misma estrategia cuando tratamos con neuronas no lineales. Para ello, utilizaremos como ejemplo una neurona sigmoidal con una función de activación logística.

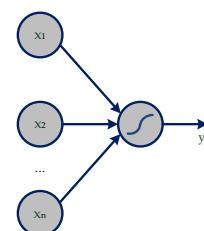


Figura 103: Una neurona sigmoidal.

La entrada neta de la neurona viene dada por la combinación lineal de los vectores de pesos y entradas, como en la neurona lineal:

$$z = w^\top x = w \cdot x = \sum_i w_i x_i$$

Aunque ya sabemos que el sesgo b de una neurona puede incluirse como un peso más $w_0 = b$ utilizando una entrada fija adicional $x_0 = 1$, en ocasiones nos encontraremos con que los sesgos se representan de forma explícita, de tal forma que la entrada neta de la neurona vendrá dada por

$$z = w^\top x + b = w \cdot x + b = \sum_{i=1}^n w_i x_i + b$$

Ahora bien, la salida de la neurona es el resultado de aplicar una función de activación no lineal a la entrada neta de la neurona:

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

En el caso de las neuronas sigmoidales, la salida de la neurona es un valor real monótonamente creciente. Conforme aumenta la entrada neta de la neurona, aumenta suavemente el valor de activación de la neurona. Además, como el valor de la función logística está acotado, la salida de la neurona estará siempre en el intervalo $[0,1]$.

Por conveniencia, evitaremos tratar explícitamente con el sesgo salvo que los detalles de implementación de un algoritmo requieran tenerlo en cuenta. Por ejemplo, la mayor parte de las herramientas utilizadas en *deep learning* emplean una matriz de pesos W y un vector de sesgos b para representar los parámetros de una capa de neuronas en una red neuronal multicapa. El funcionamiento de una capa de n entradas y m salidas se puede describir de forma vectorial mediante la expresión $y = f(Wx + b)$, donde x es un vector de n entradas, y es un vector de m salidas (con una salida para cada una de las m neuronas de la capa), W es una matriz de pesos de tamaño $m \times n$, b es un vector de sesgos [*biases*] de tamaño m y la función de activación f se aplica elemento a elemento sobre el vector resultante de evaluar la expresión matricial $Wx + b$.

La característica clave de las funciones de activación no lineales es que sean continuas y derivables para que podamos utilizar el gradiente descendente para ajustar los parámetros de una neurona no lineal, siguiendo la misma estrategia que utilizamos anteriormente para las neuronas lineales.

Para calcular el gradiente del error que nos permitirá ajustar los pesos de una neurona no lineal ($\Delta w = -\eta \nabla E$), necesitamos calcular las derivadas de la salida de la neurona con respecto a cada uno de sus pesos. Para calcular dichas derivadas, aplicamos la regla de la cadena, que nos permite representar la derivada de la salida de la neurona con respecto a un peso como el producto de las derivadas de la salida con respecto a su

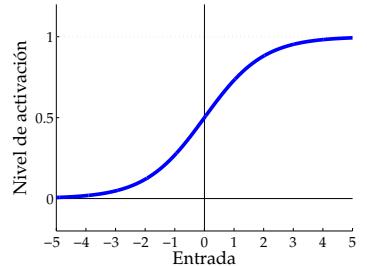


Figura 104: La función de activación de una neurona sigmoidal.

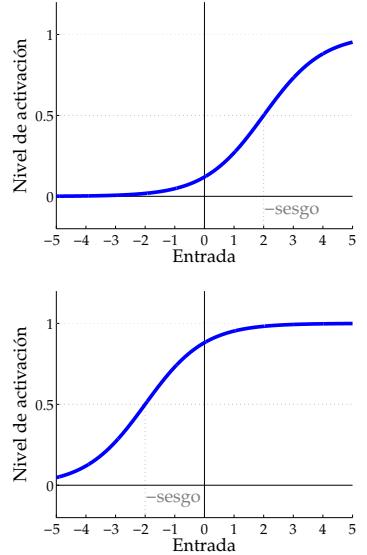


Figura 105: Efecto del sesgo [*bias*] sobre el funcionamiento de una neurona sigmoidal.

entrada neta y de su entrada neta con respecto al peso:

$$\frac{\partial y}{\partial w_i} = \frac{dy}{dz} \frac{\partial z}{\partial w_i} = f'(z) \frac{\partial z}{\partial w_i}$$

Dado que la entrada neta de la neurona es una simple combinación lineal, su derivada es muy fácil de calcular, tanto con respecto a los pesos w_i de la neurona como con respecto a sus entradas x_i :

$$\frac{\partial z}{\partial w_i} = x_i$$

$$\frac{\partial z}{\partial x_i} = w_i$$

Sólo nos falta calcular la derivada de la salida con respecto a la entrada neta. En el caso de la función logística, se trata de la derivada de la salida y con respecto al logit z , que tiene una evaluación directa en términos de la propia salida de la neurona:

$$f'(z) = \frac{dy}{dz} = y(1 - y)$$

Cuando trabajamos con la función logística, su parámetro z se conoce por el nombre de logit, ya que la función logit es la función inversa a la función logística: $\text{logit}(p) = \log(p/(1-p)) = \log(p) - \log(1-p)$

Esta derivada la podemos calcular paso a paso, para ver de dónde se obtiene la expresión anterior:

$$y = \frac{1}{1 + e^{-z}} = (1 + e^{-z})^{-1}$$

$$\frac{dy}{dz} = \frac{-1(-e^{-z})}{(1 + e^{-z})^2} = \left(\frac{1}{1 + e^{-z}}\right) \left(\frac{e^{-z}}{1 + e^{-z}}\right) = y(1 - y)$$

dado que

$$\frac{e^{-z}}{1 + e^{-z}} = \frac{(1 + e^{-z}) - 1}{1 + e^{-z}} = \frac{1 + e^{-z}}{1 + e^{-z}} + \frac{-1}{1 + e^{-z}} = 1 - y$$

Es decir, para el caso particular de la función de activación logística, combinamos los resultados anteriores en una única expresión:

$$\frac{\partial y}{\partial w_i} = \frac{dy}{dz} \frac{\partial z}{\partial w_i} = f'(z)x_i = y(1 - y)x_i$$

Esa derivada es la que utilizamos para calcular el error cometido por la neurona no lineal

$$\frac{\partial E}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial y} \frac{\partial y}{\partial w_i}$$

La derivada del error con respecto a la salida la calculamos en la sección anterior:

$$\frac{\partial E_j}{\partial y} = -(t_j - y_j)$$

Junto con la expresión que obtuvimos para la derivada de la salida no lineal con respecto al parámetro w_i , obtenemos cómo varía el error cometido por la neurona no lineal conforme cambian sus parámetros:

$$\frac{\partial E}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial w_i} = - \sum_j (t_j - y_j) y_j (1 - y_j) x_{ji}$$

En notación vectorial, el gradiente del error con respecto al vector de pesos viene dado por

$$\nabla E = - \sum_j (t_j - y_j) y_j (1 - y_j) x_j$$

Si comparamos la expresión anterior con la regla delta que obtuvimos en la sección anterior para las neuronas lineales, que era de la forma $-\sum_j (t_j - y_j) x_j$, podemos observar que la única diferencia entre ellas es la introducción del término $y(1 - y)$ correspondiente a la derivada de la función logística evaluada para su entrada neta, $f'(z) = y(1 - y)$.

Este término adicional lo único que hace es multiplicar el gradiente por la pendiente de la función de activación que estemos utilizando. Para una función de activación f cualquiera, siempre que sea derivable, el gradiente del error se puede expresar como:

$$\nabla E = - \sum_j (t_j - y_j) f'(w \cdot x_j) x_j$$

Ese gradiente lo podemos utilizar como antes, utilizando la señal de error para actualizar iterativamente los parámetros de una neurona no lineal: $\Delta w = -\eta \nabla E$.

A diferencia de lo que ocurría con el perceptrón, disponer de las derivadas del error con respecto a los parámetros de una neurona facilita el proceso de aprendizaje (i.e. el ajuste de sus parámetros). Los cambios en la salida Δy de una neurona no lineal que utiliza una función de activación continua y derivable se pueden expresar como una combinación lineal de los cambios en sus pesos Δw :

$$\Delta y \approx \sum_i \frac{\partial y}{\partial w_i} \Delta w_i = \nabla y \cdot \Delta w$$

Esta linealidad permite realizar pequeños cambios sobre los pesos de la neurona para conseguir el efecto deseado sobre su salida. Cualitativamente, las neuronas sigmoidales tienen un comportamiento similar a los perceptrones (que utilizan una función de activación escalón), pero cuantitativamente nos ofrecen un mecanismo mediante el que resulta mucho más inmediato ajustar sus parámetros.

En esta sección hemos visto cómo ampliar la regla delta para trabajar con neuronas no lineales. Para aprender los parámetros de una neurona no lineal necesitamos la derivada de su salida con respecto a cada peso.

Si multiplicamos los pesos de un perceptrón por una constante positiva, $c > 0$, su comportamiento no varía, ya que su salida depende únicamente del signo de su entrada neta, que no se ve afectado por la constante multiplicativa. En cambio, el comportamiento de las neuronas sigmoidales sí se ve afectado. De hecho, el comportamiento de la neurona sigmoidal será equivalente al de un perceptrón sólo cuando $c \rightarrow \infty$.

Si la función de activación de estas neuronas es derivable, la extensión es sencilla, pues sólo hay que aplicar la regla de la cadena para calcular el gradiente de la salida con respecto a los pesos como un producto de la derivada de la función de activación de la neurona por la derivada de su entrada neta con respecto a los pesos.

Entrenamiento de neuronas ocultas

Las redes neuronales sin unidades ocultas tienen muchas limitaciones, ya que añadir características manualmente, como hacíamos en el caso del perceptrón, resulta demasiado tedioso. Nuestro objetivo es ser capaces de entrenar redes neuronales con múltiples capas, de forma que el entrenamiento de las neuronas de sus capas ocultas les permita ser capaces de encontrar, automáticamente, características que resulten útiles en la tarea particular para la que entrenamos la red. La automatización de este proceso de extracción de características a partir del conjunto de datos de entrenamiento nos evitara tener que conocer hasta el más mínimo detalle del problema que pretendemos resolver y, lo más importante, no hará necesario un costoso proceso repetitivo de prueba y error en el que iríamos probando las características que se nos fuesen ocurriendo a ver qué tal funcionan en el problema que pretendemos resolver. La ingeniería de características se la dejamos a la red neuronal o, para ser algo más precisos, al algoritmo de entrenamiento con el que ajustaremos sus parámetros.

Utilizando la misma estrategia que empleamos para extender la regla delta para las neuronas no lineales, podemos aplicar la regla de la cadena para describir la derivada parcial del error con respecto a cada parámetro de una red neuronal multicapa como un producto de otras derivadas parciales:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

En este caso, hemos utilizado dos veces la regla de la cadena. De atrás hacia adelante:

- La derivada parcial de la entrada neta z_j de la neurona j con respecto a un peso particular w_{ij} no es más que su entrada x_i : $\frac{\partial z_j}{\partial w_{ij}} = x_i$.
- La derivada parcial de la salida y_j de la neurona j con respecto a su entrada neta z_j no es más que la derivada de la función de activación de la neurona evaluada para su entrada neta $\frac{\partial y_j}{\partial z_j} = f'(z_j)$, con $f'(z_j) = y_j(1 - y_j)$ en el caso de las neuronas sigmoidales que usan la función de activación logística. Éste es el motivo por el que el algoritmo de entrenamiento de redes multicapa requiere que las funciones de activación sean derivables.

La derivada parcial del error E con respecto a la salida y_j de la neurona es algo más problemática. En el caso de que la neurona forme parte de la capa de salida, esta derivada es fácil de evaluar (es lo que venimos haciendo desde que empezamos a utilizar la regla delta). Sin embargo, si la neurona j está en una capa oculta de la red, encontrar la derivada del error E con respecto a la salida y_j de la neurona j no resulta evidente.

En realidad, no sabemos realmente lo que debe hacer cada neurona oculta, por lo que no podemos determinar directamente cuál es el error que comete. Sin embargo, para resolver el problema, podemos utilizar la siguiente idea: aunque no sabemos qué debe hacer la neurona oculta, podemos calcular cómo cambia el error cuando cambia su actividad.

Es decir, en vez de utilizar la salida deseada para entrenar las neuronas ocultas, usamos la derivada del error con respecto a sus niveles de actividad, que es precisamente el factor que nos falta por calcular de la derivada del error con respecto a los niveles de actividad de las neuronas ocultas de la red: $\frac{\partial E}{\partial y_j}$.

Un pequeño cambio en los pesos w_{ij} de una neurona oculta j generará un cambio en su salida y_j , que podemos aproximar como

$$\Delta y_j \approx \sum_i \frac{\partial y_j}{\partial w_{ij}} \Delta w_{ij} = \nabla y_j \cdot \Delta w_j$$

De forma similar, fíjémonos en el efecto que tiene un pequeño cambio en un peso particular w_{ij} sobre el error cometido por la red, que podemos aproximar como

$$\Delta E \approx \frac{\partial E}{\partial w_{ij}^c} \Delta w_{ij}^c$$

donde hemos introducido un superíndice c para indicar que la neurona j está en la capa c de nuestra red neuronal multicapa, siendo 0 la capa de entrada de la red y C la capa de salida de la misma.

El cambio ha de propagarse desde la neurona, que está en la capa c , hasta la salida de la red, que es donde podemos evaluar directamente el error cometido. Así pues, la influencia del cambio del peso w_{ij} sobre el error, a través de un camino que conecta la neurona j con la salida de la red vendrá dada por una expresión de la forma:

$$\Delta E \approx \frac{\partial E}{\partial y_m^C} \frac{\partial y_m^C}{\partial y_n^{C-1}} \frac{\partial y_n^{C-1}}{\partial y_p^{C-2}} \cdots \frac{\partial y_q^{c+1}}{\partial y_j^c} \frac{\partial y_j^c}{\partial w_{ij}^c} \Delta w_{ij}^c$$

Ahora bien, esa expresión sólo recoge un camino particular desde la neurona j hasta la salida de la red. Para obtener el efecto conjunto que tiene el cambio del peso w_{ij} sobre el error global hemos de considerar todos los caminos posibles que conectan la neurona j con la salida de la red, de forma que

$$\Delta E \approx \sum_{mnp...q} \frac{\partial E}{\partial y_m^C} \frac{\partial y_m^C}{\partial y_n^{C-1}} \frac{\partial y_n^{C-1}}{\partial y_p^{C-2}} \cdots \frac{\partial y_q^{c+1}}{\partial y_j^c} \frac{\partial y_j^c}{\partial w_{ij}^c} \Delta w_{ij}^c$$

Una vez más, asumimos que el sesgo de la neurona viene incluido en su vector de pesos, asociado a una entrada fija $x_0 = 1$.

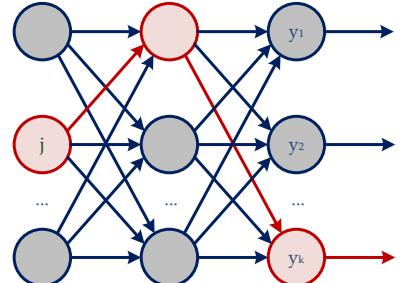


Figura 106: Uno de los caminos a través de los que un cambio en la actividad de la neurona j afecta a la salida de la red.

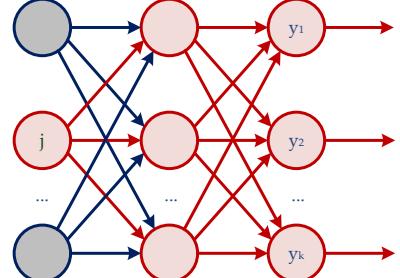


Figura 107: Todos los caminos a través de los que un cambio en la actividad de la neurona j puede afectar a la salida de la red multicapa.

Sustituyendo en la expresión inicial que relacionaba ΔE con Δw_{ij}^c , obtenemos la derivada del error E con respecto al peso w_{ij}^c :

$$\frac{\partial E}{\partial w_{ij}^c} = \sum_{mnp...q} \frac{\partial E}{\partial y_m^C} \frac{\partial y_m^C}{\partial y_n^{C-1}} \frac{\partial y_n^{C-1}}{\partial y_p^{C-2}} \cdots \frac{\partial y_q^{c+1}}{\partial y_j^c} \frac{\partial y_j^c}{\partial w_{ij}^c}$$

La expresión resultante puede parecer complicada, incluso monstruosa, pero tiene una estructura clara que podemos interpretar de forma intuitiva. Nuestro objetivo es descubrir cómo modificar los parámetros de la red de forma que se minimice el error y la expresión anterior nos dice cómo cambia el error E conforme variamos un parámetro individual w_{ij} .

De hecho, la variación del error para todos los pesos w_{ij} de una neurona particular j tiene la misma estructura, por lo que, de la expresión anterior, podemos extraer la siguiente:

$$\frac{\partial E}{\partial y_j^c} = \sum_{mnp...q} \frac{\partial E}{\partial y_m^C} \frac{\partial y_m^C}{\partial y_n^{C-1}} \frac{\partial y_n^{C-1}}{\partial y_p^{C-2}} \cdots \frac{\partial y_q^{c+1}}{\partial y_j^c}$$

El error se ve afectado por la actividad y_j de una neurona oculta j de la capa c y a él contribuyen todos los caminos que conectan la salida de la neurona j con la salida de la red. Por este motivo, la derivada del error es una suma sobre todos los caminos posibles. Para cada camino individual desde la neurona j hasta la salida de la red, simplemente se van multiplicando las derivadas parciales de los niveles de activación de cada neurona con respecto al nivel de activación de la neurona que la precede en ese camino hacia la salida.

El algoritmo de propagación de errores hacia atrás, que utilizaremos para ajustar los parámetros de una red multicapa, lo único que hace es calcular de una forma eficiente esa suma de las contribuciones de todos los caminos desde una neurona hasta la salida. Para ello, aprovecha la topología de la red, organizada en una serie de capas con conexiones única y exclusivamente entre capas adyacentes. Esa topología le permite realizar el cálculo del gradiente de forma inteligente, con lo que se puede determinar eficientemente cómo se propagan a través de la red pequeñas perturbaciones en los pesos de las neuronas ocultas, cómo alcanzan la salida esas perturbaciones y cómo afectan a la función de error.

En resumen, de nuestro análisis preliminar sobre el entrenamiento de neuronas ocultas, deberíamos extraer las siguientes conclusiones:

- La actividad de cada neurona oculta puede tener efectos en muchas neuronas de salida, por lo que debemos combinarlos. Para ello, haremos un seguimiento de los efectos de la actividad de una neurona oculta a través de todos los caminos que la conectan con la salida de la red.
- Una vez que obtenemos las derivadas del error con respecto a los niveles de actividad de las neuronas ocultas, $\frac{\partial E}{\partial y_j^c}$, podemos calcular

las derivadas del error para sus pesos de entrada, $\frac{\partial E}{\partial w_{ij}}$, que es lo que realmente nos interesa. Para ello, sólo tenemos que utilizar la expresión con la que abrimos este apartado:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

El algoritmo de propagación de errores hacia atrás

Llegados a este punto, es el momento de completar el algoritmo que nos permitirá entrenar redes neuronales multicapa. Técnicamente, el término *backpropagation* se refiere únicamente al método mediante el que calculamos el gradiente necesario para ajustar los parámetros de la red. Recordemos que, al final, lo que pretendemos es aplicar iterativamente un método de optimización basado en el gradiente descendente para ir ajustando los pesos de la red neuronal y conseguir entrenarla para que haga lo que queramos:

$$\Delta w = -\eta \nabla_w E$$

Para las neuronas de la capa de salida, ese gradiente es fácil de calcular, basta con utilizar la regla delta. Sólo necesitamos definir una función de error, coste o pérdida y calcular su derivada con respecto a los pesos de las neuronas de salida para poder ajustar dichos pesos. Por regla general, utilizaremos una función de error que cumpla dos propiedades:

- La función de error se puede especificar como un promedio de las funciones de coste asociadas a ejemplos concretos del conjunto de entrenamiento. Esto nos permitirá realizar una estimación instantánea del gradiente para cada ejemplo particular y calcular el gradiente para el conjunto de entrenamiento en su conjunto:

$$E = \frac{1}{n} \sum_j E_j$$

$$\nabla_y E = \frac{1}{n} \sum_j \nabla E_j$$

- La función de coste asociada a cada ejemplo del conjunto de entrenamiento la definimos en términos de la salida que se obtiene de la red y_j y del valor que deseamos obtener t_j . Por ejemplo, si utilizamos el error cuadrático:

$$E_j = \frac{1}{2} (t_j - y_j)^2$$

$$\nabla_y E_j = -(t_j - y_j)$$

Para calcular el gradiente del error, obviamente, primero tenemos que calcular la salida de la red a partir de un ejemplo del conjunto de entrenamiento. Por este motivo, el algoritmo de aprendizaje basado en *backpropagation* y gradiente descendente consta, pues, de dos fases:

- Una fase de propagación hacia adelante, en la que le suministramos a la red un patrón de entrada y calculamos la salida de la red para dicho patrón.
- Una fase de propagación hacia atrás, de donde proviene el término *backpropagation*, en la que evaluamos el error cometido por la red y propagamos dicho error hacia atrás, capa por capa, de forma que se pueda calcular eficientemente el gradiente del error para las neuronas ocultas de la red.

En el caso de las neuronas de salida, y^C , podemos calcular el gradiente del error con respecto al nivel de activación de la neurona directamente a partir de las salidas observadas y_j y de las salidas deseadas t_j :

- Si usamos aprendizaje por lotes:

$$\nabla_y E = - \sum_j (t_j - y_j)$$

- Si utilizamos aprendizaje *online*:

$$\nabla_y E = -(t_j - y_j)$$

El error en la salida de una neurona puede deberse a los valores de sus pesos, a las entradas que se reciben de las capas anteriores de la red o a una combinación de ambos factores, motivo por el que usaremos la entrada neta z_j de la neurona como base para nuestros cálculos, donde $z_j = w_j \cdot x_j$. Para ello, definimos δ_j^c para una neurona j de la capa c como la derivada parcial del error con respecto a la entrada neta z_j^c de la neurona.

$$\delta_j^c = \frac{\partial E}{\partial z_j^c}$$

Intuitivamente, este “delta” nos indica cómo influye una perturbación en la entrada neta de la neurona Δz_j^c en el error de la red. La perturbación en la entrada neta de la neurona, que puede deberse a un cambio en alguno de sus pesos o en alguna de sus entradas, modificará la salida y_j^c de la neurona, que pasará de $f(z_j^c)$ a $f(z_j^c + \Delta z_j^c)$. A su vez, este cambio en el nivel de activación de la neurona ocasionará un cambio en el error de la red, que podemos aproximar por $\frac{\partial E}{\partial z_j^c} \Delta z_j^c$. Ese error es el que intentaremos corregir reduciendo la entrada neta de la neurona en sentido opuesto a $\frac{\partial E}{\partial z_j^c}$.

Al pasar de las derivadas parciales con respecto a la salida, $\frac{\partial E}{\partial y}$, a las derivadas parciales con respecto a la entrada neta, $\frac{\partial E}{\partial z}$, lo que hemos hecho es incorporar la función de activación f de la neurona en el cálculo de los “deltas” que reutilizaremos en el cálculo del gradiente del error:

$$\delta_j^c = \frac{\partial E}{\partial z_j^c} = \frac{\partial E}{\partial y_j^c} \frac{dy_j^c}{dz_j^c} = \frac{\partial E}{\partial y_j^c} f'(z_j^c)$$

donde f' es la derivada de la función de activación de la neurona de salida: 1 para una neurona lineal, $y_j(1 - y_j)$ para la función logística, $(1 + y_j)(1 - y_j)$ para la tangente hiperbólica y $1_{w \cdot x_j \geq 0}$ para las unidades lineales rectificadas (ReLU).

¿Cómo calculamos de forma eficiente las derivadas del error para las neuronas de las capas ocultas?

Para calcular el gradiente del error con respecto a los parámetros de las neuronas ocultas de la red, ya vimos que teníamos que considerar todos los caminos que conectan la salida de una neurona oculta con las salidas de la red neuronal. Ahora bien, en vez de ir enumerando todos esos caminos uno por uno, aprovecharemos la topología de la red para hacer ese cálculo de una forma mucho más eficiente.

La derivada parcial del error con respecto al nivel de activación de una neurona oculta era de la forma:

$$\frac{\partial E}{\partial y_j^c} = \sum_{mnp...q} \frac{\partial E}{\partial y_m^C} \frac{\partial y_m^C}{\partial y_n^{C-1}} \frac{\partial y_n^{C-1}}{\partial y_p^{C-2}} \cdots \frac{\partial y_q^{c+1}}{\partial y_j^c}$$

donde vemos que la derivada del error se calcula sumando las contribuciones de distintos caminos. Ahora bien, dado que las distintas neuronas de la capa c comparten los caminos que van desde la capa $c + 1$ hasta la salida de la red, esto nos permitirá calcular el gradiente del error para la capa $c + 1$ y, a partir de ese gradiente, calcular el gradiente para las neuronas de la capa c .

Sólo tenemos que observar cómo se relaciona el error de las neuronas de una capa oculta con el error de las neuronas de la siguiente capa. Para ello, expresamos la contribución al error de las neuronas de la capa c en términos de la contribución al error de las neuronas de la capa $c + 1$:

$$\frac{\partial E}{\partial y_j^c} = \sum_k \frac{\partial E}{\partial y_k^{c+1}} \frac{\partial y_k^{c+1}}{\partial y_j^c}$$

A continuación, reescribimos la expresión anterior en términos de las entradas netas de las neuronas, en vez de emplear sus niveles de activación:

$$\begin{aligned}
\delta_j^c &= \frac{\partial E}{\partial z_j^c} \\
&= \frac{\partial E}{\partial y_j^c} f'(z_j^c) \\
&= \left(\sum_k \frac{\partial E}{\partial y_k^{c+1}} \frac{\partial y_k^{c+1}}{\partial y_j^c} \right) f'(z_j^c) \\
&= \left(\sum_k \left(\frac{\partial E}{\partial y_k^{c+1}} \frac{\partial y_k^{c+1}}{\partial z_j^{c+1}} \right) \frac{\partial z_k^{c+1}}{\partial y_j^c} \right) f'(z_j^c) \\
&= \left(\sum_k \frac{\partial E}{\partial z_j^{c+1}} \frac{\partial z_k^{c+1}}{\partial y_j^c} \right) f'(z_j^c) \\
&= \left(\sum_k \delta_j^{c+1} \frac{\partial z_k^{c+1}}{\partial y_j^c} \right) f'(z_j^c)
\end{aligned}$$

Ahora bien, la salida de la capa c es la entrada de la capa $c + 1$, $y^c = x^{c+1}$, por lo que podemos sustituir y^c por x^{c+1} en la expresión anterior y calcular las derivadas parciales de la entrada neta z^{c+1} con respecto a las entradas individuales x^{c+1} :

$$\begin{aligned}
\delta_j^c &= \left(\sum_k \delta_j^{c+1} \frac{\partial z_k^{c+1}}{\partial x_j^{c+1}} \right) f'(z_j^c) \\
&= \left(\sum_k \delta_j^{c+1} w_{jk}^{c+1} \right) f'(z_j^c)
\end{aligned}$$

Hemos obtenido una expresión que nos permite calcular los deltas de la capa c a partir de los deltas de la capa $c + 1$. En notación vectorial:

$$\delta^c = ((W^{c+1})^\top \delta^{c+1}) \odot f'(z^c)$$

donde \odot es el producto elemento a elemento de dos vectores (conocido formalmente como producto Hadamard o producto Schur), de tal forma que los elementos del vector resultante de $v \odot w$ son $[v \odot w]_j = v_j w_j$

Usando el vector de “deltas” δ^c para las neuronas de la capa c , podemos obtener todos los valores que necesitamos para completar el cálculo del gradiente del error con respecto a todos los parámetros de la red. El algoritmo de propagación de errores, *backpropagation*, queda como sigue:

1. En primer lugar, para la capa de salida C , calculamos

$$\delta^C = \nabla_y E \odot f'(z^C)$$

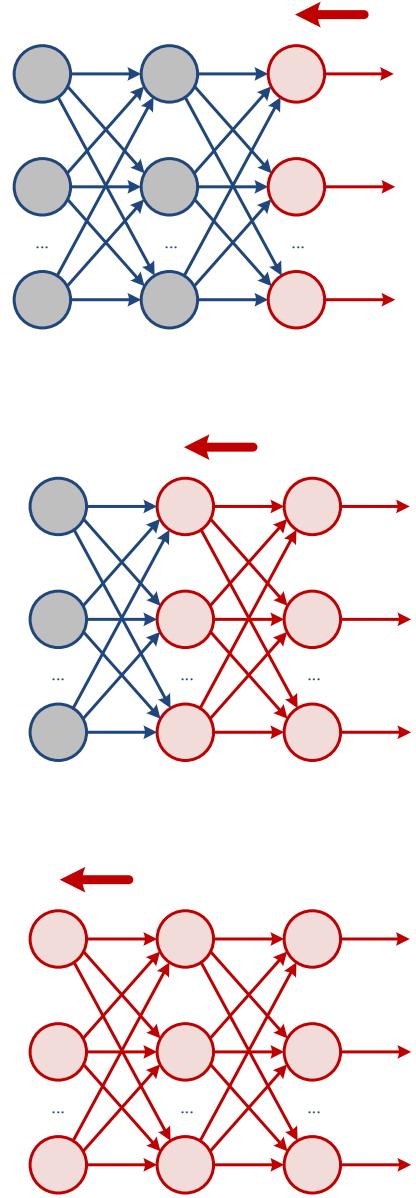


Figura 108: Cálculo del gradiente usando propagación de errores hacia atrás (*backpropagation*).

2. A continuación, vamos calculando los “deltas” para las capas ocultas de la red, obteniendo los deltas de la capa c a partir de los deltas de la capa $c + 1$:

$$\delta^c = ((W^{c+1})^\top \delta^{c+1}) \odot f'(z^c)$$

Combinando las dos expresiones anteriores podemos calcular el gradiente del error para todas las capas de la red multicapa, empezando por la de salida. La primera expresión nos permite calcular el gradiente δ^C para las neuronas de la capa de salida. A partir de él, utilizamos la segunda expresión para calcular el gradiente δ^{C-1} de la capa $C - 1$ a partir de δ^C , δ^{C-2} a partir de δ^{C-1} ... y así sucesivamente hasta llegar al gradiente δ^1 la primera capa de la red.

3. Una vez que tenemos todos los “deltas”, podemos calcular fácilmente cómo varía el error en función de los pesos de las neuronas. Teniendo en cuenta que la salida de la capa $c - 1$ es la entrada de la capa c ($x^c = y^{c-1}$), la derivada del error con respecto a los pesos la podemos expresar como:

$$\frac{\partial E}{\partial w_{ij}^c} = \frac{\partial E}{\partial z_j^c} \frac{\partial z_j^c}{\partial w_{ij}^c} = \delta_j^c x_i^c = \delta_j^c y_i^{c-1}$$

Si en nuestra implementación tratamos el sesgo b_j como un parámetro separado del resto de los pesos de una neurona, sólo tenemos que acordarnos de que el sesgo equivale a un peso asociado a una entrada fija $x_0 = 1$, por lo que

$$\frac{\partial E}{\partial b_j^c} = \delta_j^c$$

En definitiva, *backpropagation* se encarga de evaluar de forma eficiente la contribución de cada neurona oculta al error observado en la capa de salida. Para ello, se aprovecha de que en una red organizada por capas, la propagación de los niveles de activación de las neuronas de la capa c utilizan los mismos caminos para llegar a la salida de la red y todos ellos pasan por la capa $c + 1$.

El algoritmo de propagación de errores nos proporciona un método eficiente para calcular las derivadas del error $\frac{\partial E}{\partial w_{ij}}$ para cada peso de una red multicapa. Prescindiendo de los superíndices que utilizamos para indicar la capa en la que se encontraba cada neurona, cada una de las capas de la red puede estimar el gradiente del error con respecto a su matriz de pesos como el producto exterior [*outer product*] de su vector de deltas δ y su vector de entradas x :

$$\nabla_W E = x \otimes \delta$$

donde el producto exterior $x \otimes \delta$ de un vector x de tamaño n y un vector δ de tamaño m es una matriz $\nabla_W E$ de tamaño $n \times m$ en la que su elemento (i, j) se define como $[\nabla_W E]_{ij} = x_i \delta_j$.

El producto exterior es el nombre que recibe el producto tensorial de dos vectores, que genera como resultado una matriz, a diferencia del producto interno, más conocido como producto escalar, ya que genera un escalar. El producto exterior es un caso particular del producto de Kronecker.

Del mismo modo, el gradiente del error con respecto al vector de sesgos de las neuronas de una capa es, simplemente,

$$\nabla_b E = \delta$$

Las expresiones anteriores nos permiten calcular la derivada parcial de la función de error con respecto a todos los parámetros de una red multicapa (pesos y sesgos), indicándonos cómo cambia el error de la red cuando cambian los valores de sus parámetros.

Una vez que tenemos los gradientes del error con respecto a los parámetros de la red, como resultado de la propagación de errores hacia atrás, podemos aplicar una regla de actualización de los parámetros de la red similar a la regla delta. Por este motivo, la regla utilizada en el entrenamiento de redes multicapa se suele denominar regla delta generalizada:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

donde η es la tasa de aprendizaje utilizada por el método de gradiente descendente. Como en la regla delta, modificamos los pesos en la dirección de máxima pendiente dada por el gradiente de la función de error. Como queremos minimizar el error, lo hacemos en sentido opuesto al indicado por el gradiente.

En el entrenamiento de redes multicapa, para actualizar el peso asociado a cada sinapsis de la red, multiplicamos su delta de salida por su valor de entrada para obtener el gradiente del error con respecto al peso. A continuación, se le resta una fracción de ese peso, dada por la tasa de aprendizaje.

La tasa de aprendizaje determina la velocidad de convergencia del algoritmo. Cuanto mayor sea, más rápido se entrena la red, si bien corremos el riesgo de que la solución oscile y no termine de converger (e incluso diverja). Cuanto menor sea la tasa de aprendizaje, más preciso será el entrenamiento de la red, pero también puede requerir mucho más tiempo. Por estos motivos, es habitual utilizar tasas de aprendizaje adaptativas, con valores elevados al comienzo del entrenamiento, de forma que nos movamos rápidamente durante las primeras iteraciones, y valores más pequeños al final del entrenamiento, para afinar en nuestra búsqueda del mínimo de la función de coste.

Recapitulación, a modo de receta

Asumiendo que queremos entrenar una red multicapa utilizando un algoritmo de aprendizaje *online* y una función de error cuadrática, tipo MSE o SSE, veremos cómo se aplica el algoritmo de propagación de errores combinado con un método de optimización basado en el gradiente descendente estocástico.

En primer lugar, comenzamos por calcular los deltas asociados a las diferentes neuronas de la red, para lo que recorremos la red hacia atrás (desde la capa de salida hasta la capa de entrada):

$$\delta_j^k = \begin{cases} f'(z_j^k) (y_j - t_j) & \text{en las neuronas de salida,} \\ f'(z_j^k) \left(\sum_p \delta_p^{k+1} w_{pj}^{k+1} \right) & \text{en las neuronas ocultas.} \end{cases}$$

donde δ_j^{k+1} son valores ya calculados de δ_j para las neuronas de la siguiente capa de la red neuronal.

Una vez realizada la propagación hacia atrás de los errores, procedemos a actualizar los pesos de la red utilizando la expresión asociada al gradiente descendente estocástico:

$$\Delta w_{ij}^k = -\eta x_i^k \delta_j^k$$

Para ser más explícitos, cada peso se actualiza aplicando la siguiente expresión:

$$\Delta w_{ij}^k = \begin{cases} -\eta x_i^k f'(z_j^k) (y_j - t_j) & \text{en las neuronas de salida,} \\ -\eta x_i^k f'(z_j^k) \left(\sum_p \delta_p^{k+1} w_{pj}^{k+1} \right) & \text{en las neuronas ocultas.} \end{cases}$$

En las expresiones anteriores aparece la derivada de la función de activación $f'(z_j)$. Si empleamos la función logística, $f'(z_j) = y_j(1 - y_j)$. En el ajuste de un peso asociado a la entrada x_i de una neurona, por tanto, intervienen tres factores: el valor de la entrada x_i , la derivada de su función de activación $f'(z_j)$ dada su entrada neta y una señal de error que depende de la capa en la que nos encontramos y que proviene de las siguientes capas de la red.

El gradiente en las distintas capas de la red

En el algoritmo de propagación de errores, partimos de una señal de error observada en la capa de salida de la red y vamos propagando esa señal de error hacia atrás por toda la red.

En vez de utilizar la derivada del error con respecto a la salida de una neurona, $\frac{\partial E}{\partial y_j}$, por conveniencia utilizamos la derivada del error con respecto a la entrada neta de la neurona, $\frac{\partial E}{\partial z_j}$, a la que llamamos delta δ_j :

$$\delta_j = \frac{\partial E}{\partial y_j} f'(z_j)$$

Pudimos observar que el gradiente del error de una neurona oculta se podía calcular combinando los gradientes del error para las neuronas de la siguiente capa de la red:

$$\delta^c = f'(z^c) \odot ((W^{c+1})^\top \delta^{c+1})$$

Podemos sustituir el producto Hadamard \odot de la expresión anterior si creamos una matriz diagonal D que contenga, en su diagonal, los valores de las derivadas de las funciones activación para las distintas neuronas de la capa, $f'(z^c)$:

$$\delta^c = D^c (W^{c+1})^\top \delta^{c+1}$$

Dado el producto matricial anterior, cuando tenemos múltiples capas ocultas, los deltas podemos desarrollarlos como

$$\begin{aligned} \delta^c &= D^c (W^{c+1})^\top \dots D^{C-1} (W^C)^\top D^C \nabla_y E \\ &= \left(\prod_{k=c}^{C-1} D^k (W^{k+1})^\top \right) D^C \nabla_y E \end{aligned}$$

Dada esa expresión, podemos analizar cómo contribuye el error observado en la capa de salida, $\nabla_y E$, al gradiente del error en una neurona oculta:

- Cuando una señal de entrada a una neurona es pequeña, el término asociado al gradiente del error con respecto al peso de la sinapsis asociada a esa entrada también será pequeño, por lo que el peso no cambiará demasiado durante el gradiente descendente ($\nabla_W E = x \otimes \delta$). Dicho de otra forma, si el nivel de activación de una neurona es bajo, los pesos asociados a las sinapsis que reciben como entrada la salida de la neurona se ajustarán lentamente.
- La presencia de pesos elevados en los caminos desde la neurona oculta hasta la salida indicarán una contribución elevada al gradiente del error... salvo que las derivadas de las funciones de activación de las neuronas, reflejadas en las matrices D , sean pequeñas. Si utilizamos funciones de activación sigmoidales, cuando una neurona se satura, en uno u otro sentido, la función sigmoidal es prácticamente plana, por lo que su derivada es aproximadamente cero ($\sigma'(z^c) \approx 0$). En el momento en el que una neurona sigmoidal se satura, los gradientes del error serán muy bajos y la neurona oculta dejará de aprender, ya que los ajustes que realizamos sobre sus pesos son proporcionales a la derivada de su función de activación. En el mejor de los casos, aprenderá muy lentamente.
- Cuando tenemos múltiples capas ocultas en una red multicapa, la expresión de los gradientes δ^c incluye múltiples factores de la forma $D^k (W^{k+1})^\top$. Como el número de factores de este tipo depende de la profundidad a la que se encuentra cada neurona, eso hará que las neuronas de capas diferentes aprendan a ritmos diferentes.

Lo más habitual es que las capas más alejadas de la salida de la red aprendan más lentamente que las capas más cercanas a la salida. Dado que los niveles de activación de las neuronas están acotados y

las neuronas sigmoidales tienden a saturarse, los factores resultantes a menudo son menores que 1. En casos extremos, el gradiente será prácticamente nulo, por lo que la red será incapaz de aprender. Es un problema conocido como desaparición del gradiente o gradiente evanescente [*vanishing gradient*].

También puede darse el caso opuesto. Si los pesos de la red aumentan en exceso durante el entrenamiento de la red (p.ej. si utilizamos una tasa de aprendizaje demasiado elevada que hace que el gradiente descendente sea inestable y diverja), los factores involucrados en el cálculo del gradiente tomarán valores mucho mayores que 1. En este caso, el problema es el contrario al gradiente evanescente y hablaríamos de la explosión del gradiente [*exploding gradient*].

En resumen, el aprendizaje basado en gradiente descendente usando *backpropagation* puede no funcionar demasiado bien cuando el nivel de activación de una neurona j es bajo (lo que afecta al ajuste de los pesos de las neuronas que reciben como entrada la salida de la neurona j), las neuronas sigmoidales se saturan (por lo que las derivadas de sus funciones de activación serán bajas, lo cual afecta al entrenamiento de sus pesos y, potencialmente, al de las neuronas que las preceden en la red) o la profundidad de nuestra red hace que se vea afectada por problemas como la desaparición o la explosión del gradiente.

En esta sección hemos visto cómo se puede calcular el gradiente del error para los parámetros de una red neuronal multicapa, utilizando un algoritmo de propagación de errores hacia atrás denominado *backpropagation*. En combinación con técnicas de optimización basadas en el gradiente descendente, nos proporciona un algoritmo con el que entrenar redes neuronales de múltiples capas. Este algoritmo es la base de todos los éxitos recientes del *deep learning*, de ahí que lo hayamos analizado con detenimiento.

La regla de ajuste de los pesos de una red derivada del uso de *backpropagation* se denomina regla delta generalizada. Dado que incorpora la derivada de la función de activación, este hecho limita los cambios que se realizan sobre los pesos y facilita la estabilidad del algoritmo de aprendizaje. El requisito que impone es, obviamente, que la función de activación que utilicemos para las neuronas de nuestra red ha de ser derivable.

A fin de cuentas, *backpropagation*, la piedra angular del *deep learning*, no es más que el cálculo del gradiente de una función de error, coste o pérdida. Este cálculo se realiza aplicando repetidamente la regla de la cadena y, desde el punto de vista matemático, se traduce en multiplicar jacobianos por gradientes. El resto, sólo detalles...

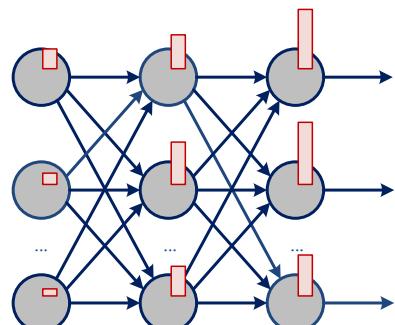


Figura 109: La magnitud del gradiente varía de una capa a otra durante el entrenamiento de una red neuronal multicapa.

Implementación del algoritmo

Veamos, paso a paso, cómo conseguir una implementación funcional del algoritmo de propagación de errores hacia atrás...

Funciones de activación

El primer paso de nuestra implementación consiste en modelar distintas funciones de activación. Como queremos disponer de distintas funciones de activación, podemos crear una clase abstracta que defina dos operaciones: `eval` para evaluar la función de activación y `diff` para evaluar su derivada.

```
class Activation:
    function y = eval(z);
    function d = diff(z);
```

Utilizando esta clase base, creamos subclases para representar las funciones de activación más comunes:

- Función logística:

```
class Sigmoid: Activation

function y = eval(z)
    y = 1.0 / (1.0 + exp(-z));

function d = diff(z)
    y = eval(z);
    d = y * (1-y);
```

- Tangente hiperbólica:

```
class TANH: Activation

function y = eval(z)
    e2z = exp(2*z);
    y = (e2z-1) / (e2z+1);

function d = diff(z)
    y = eval(z);
    d = 1 - y*y;
```

- Unidad lineal rectificada, ReLU [*Rectifier Linear Unit*]:

```

class RELU: Activation

    function y = eval(z)
        y = (z>=0)? z: 0;

    function d = diff(z)
        d = (z>=0)? 1: 0;

```

El segundo paso de nuestra implementación consistiría en modelar las neuronas individuales de nuestra red neuronal. Para ello, asumiremos que cada neurona tiene un conjunto de parámetros que consta de un vector de pesos w [*weights*] y un sesgo b [*bias*], además de una función de activación *activation* de tipo *Activation*. Para una neurona dada, podríamos definir las siguientes operaciones:

- Calcular la entrada neta de la neurona para una entrada x :

$$z = w \cdot x + b$$

```

function z = net(x)
    z = b;
    for i=1:size(w)
        z += w[i]*x[i];

```

- Calcular la salida de la neurona dada una entrada x :

$$y = f(w \cdot x + b)$$

```

function y = output(x)
    z = net(x);
    y = activation(z);

```

Propagación hacia adelante

Normalmente, no tendremos una única neurona en cada capa de la red neuronal, por lo que los cálculos anteriores tendríamos que repetirlos para todas y cada una de las neuronas de la capa con la que estemos trabajando.

Resulta más adecuado modelar cada capa como un objeto que disponga de una matriz de pesos w y un vector de sesgos b . Asumimos que la función de activación es la misma para todas las neuronas de la misma capa. Así pues, comenzamos nuestra definición de la clase **Layer**, que utilizaremos para modelar una capa completa de la red neuronal, con n entradas y m salidas. Por ahora, nos basta con definir un método **forward** que se encargue de obtener la salida de una capa de la red a partir de la entrada x que le proporcionemos:

```
class Layer(n,m):
    function y = forward(x)
        for j=1:m
            z = b[j]
            for i=1:n
                z += w[j][i]*x[i];
            y[j] = activation.eval(z);
```

Dependiendo de la plataforma de programación que utilicemos, los arrays puede que comiencen con el índice 0 (C, C++, Java, C#, Python, Ruby, Perl, Go, Swift, Scala, Haskell, Visual Basic) o 1 (Matlab, Mathematica, Julia, Lua, Fortran, Pascal, Delphi, AWK). Tendremos que adaptar el pseudocódigo que aquí aparece a los detalles concretos de nuestro lenguaje favorito, teniendo cuidado con los índices.

Además, en nuestra implementación particular puede que el sesgo, en vez de tenerlo separado del resto de los pesos, lo modelemos como un peso más (w_0 , asociado a una entrada fija $x_0 = 1$), por lo que en el fragmento de código anterior utilizaríamos $w[j][0]$ en lugar de $b[j]$.

Observe, por otro lado, la forma en la que hemos dispuesto la matriz de pesos w . Utilizamos la salida j como primera dimensión del array para mejorar la localidad espacial de los accesos a la matriz, lo que nos permitirá aprovechar mejor la jerarquía de memoria de nuestro ordenador, ya que los accesos a la caché son mucho más rápidos que los accesos a memoria principal y, con un poco de suerte, la primera vez que accedamos a una fila del array cargaremos en caché la fila completa.

La clase `Layer` nos sirve para modelar una capa completa de una red neuronal multicapa. Una red neuronal multicapa de tipo *feed-forward* no será más que un array `layer` de capas de tipo `Layer`. Para calcular la salida de la red, definimos la función que nos permite propagar una señal desde la capa de entrada hasta la capa de salida:

```
class FeedForwardNetwork:
    function y = forward(x)
        y = x;
        for k=1:size(layer)
            y = layer[k].forward(y);
```

Así pues, ya sabemos cómo obtener la salida de una red multicapa a partir de una entrada cualquiera en forma de vector x . Pasemos ahora a la parte realmente interesante, la que nos permitirá entrenar la red multicapa ajustando sus pesos.

Función de error

Normalmente, dispondremos de un conjunto de datos de entrenamiento que contenga múltiples parejas de vectores de entrada y salida. Nuestra

intención es que la red neuronal sea capaz de, dada una entrada concreta como las del conjunto de entrenamiento, generar una salida como la que aparece asociada a dicha entrada en el conjunto de entrenamiento. Además, nos gustaría que la red fuese capaz de generalizar adecuadamente para vectores de entrada distintos a los que están presentes en el conjunto de entrenamiento.

A partir de ahora, supondremos que disponemos de conjuntos de datos modelados por medio de una clase `Dataset`. Un conjunto de entrenamiento estará formado por patrones (`data[i], target[i]`), que podremos obtener directamente de un objeto de tipo `Dataset`. Tanto `data` como `target` serán matrices en las que cada fila corresponderá a un vector de características (`data` para las entradas y `target` para las salidas deseadas). El tamaño de nuestro conjunto de datos lo podremos consultar llamando a un método `size` de la clase `Dataset`.

Para evaluar el comportamiento de una red neuronal sobre un conjunto de datos, definiremos una función de error que compare las salidas generadas por la red (usando el algoritmo implementado por el método `forward` de la sección anterior) con las salidas que deseamos obtener. Por ejemplo, podemos calcular la suma de los errores al cuadrado para los distintos ejemplos de un conjunto de datos:

```
function error = SSE (net, set)
    error = 0;
    for k=1:set.size()
        output = net.forward(set.data[k]);
        for i=1:size(output)
            error += (set.target[k][i] - output[i])^2;
```

A partir de esta función, resulta trivial calcular el error cuadrático medio:

```
function error = MSE (net, set)
    error = SSE(net, set) / set.size();
```

Ésta es la función que se minimiza al aplicar la regla delta generalizada en una red neuronal multicapa entrenada con backpropagation.

Propagación hacia atrás

Una vez que hemos calculado la salida de la red para un conjunto de datos de entrada, evaluamos una función de error que nos permita comparar la salida obtenida (`y`) con la salida deseada (`[t]arget`). Para minimizar esa función de error, ajustaremos iterativamente los pesos de la red de acuerdo al gradiente del error con respecto a los pesos.

Formalmente, tenemos que calcular cuánto cambia el error con respecto a cada peso: las derivadas parciales $\partial E / \partial w_{ij}$, que podemos representar

Normalmente, se suele incluir un factor $1/2$ en la definición de la función de error cuadrático, que desaparecerá al derivar la función de error y nos evitirá tener que ir arrastrando una constante en cálculos posteriores.

vectorialmente en forma de gradiente $\nabla_W E$. Una vez que dispongamos de estas derivadas parciales, modificaremos los pesos ligeramente en la dirección que más reduce el error usando la tasa de aprendizaje η , utilizando un método de optimización basado en el gradiente descendente.

La idea clave del algoritmo de propagación de errores *backpropagation* consiste en transmitir la señal de error desde la capa de salida hasta cada nodo intermedio que ha contribuido a la salida de la red. Cada nodo recibe una parte de la señal de error, en proporción a su contribución relativa a la salida original. Como vimos anteriormente, este cálculo se hace de forma eficiente propagando el error por capas, hacia atrás, desde la capa de salida hasta la capa de entrada.

En primer lugar, calculamos el error para cada neurona de la capa de salida. Utilizando aprendizaje *online*, realizamos una estimación instantánea del gradiente para un ejemplo particular del conjunto de datos:

```
function error = error (target, output)
    for i=1:size(output)
        error[i] = (target[i] - output[i]);
```

Ese vector de error lo utilizamos para comenzar el cálculo de los “deltas” por capas, empezando por la capa de salida:

```
class FeedForwardNetwork:
    function backward(error)
        layers = size(layer)
        for j=1:size(error)
            delta[layers][j] = error[j] * activation.diff(layer[layers].z[j]);
        ...
    
```

donde hemos utilizado la entrada neta de las neuronas de la capa de salida para evaluar la derivada de su función de activación, que multiplicamos por el error para obtener los deltas δ_j^k asociados a la capa de salida.

A continuación, propagamos los deltas de la capa $k + 1$ a la capa k , lo que podemos implementar mediante el siguiente bucle:

```
...
for k=layers-1:1
    for j=1:layer[k].outputs()
        error[k][j] = dot( layer[k+1].w[j], delta[k+1][j] );
        delta[k][j] = error[k][j] * activation.diff(layer[k].z[j]);
```

donde `dot(w,delta)` realiza el producto escalar de los vectores de pesos y deltas de la siguiente capa de la red, que previamente hemos calculado. La implementación de esta función, en caso de que no dispongamos de ella, es trivial:

```
function dot (x,y)
    dot = 0;
    for i=0:size(x)
        dot += x[i]*y[i];
```

Ya hemos calculado los deltas δ_j^k asociados a todas las neuronas de la red, lo que nos permite calcular las derivadas del error con respecto a todos los parámetros de la red y la actualización de los mismos utilizando el método del gradiente descendente.

Actualización de los pesos de la red

La propagación hacia atrás del error nos permitió calcular los deltas δ_j^k , que en nuestra clase `FeedForwardNetwork` hemos guardado en una matriz `delta`. Una vez que tenemos las derivadas de error con respecto a las entradas netas de todas las neuronas de la red, podemos actualizar los pesos y sesgos de las neuronas de forma casi trivial.

A partir de ellos, calcular las derivadas del error con respecto a los pesos de la red resulta sencillo, ya que $\partial E / \partial w_{ij}^k = x_i^k * \delta_j^k$ y ése es el valor que tenemos que utilizar para actualizar los pesos de la red. De forma similar, las derivadas del error con respecto a los sesgos, que recordemos equivalen a pesos asociados a una entrada fija $x_0 = 1$, ya los tenemos calculados: $\partial E / \partial b_j^k = \delta_j^k$

Para concluir nuestra implementación inicial, definimos un tercer método para la clase `FeedForwardNetwork`. Este método se encarga de actualizar los parámetros de toda la red utilizando una tasa de aprendizaje `eta`:

```
class FeedForwardNetwork:
    function y = update(eta)
        for k=1:layers
            for j=1:layer[k].outputs()
                layer[k].b[j] += eta * delta[k][j];
            for i=1:layer[k].inputs()
                layer[k].w[j][i] += eta * layer[k].input[i] * delta[k][j];
```

Observe cómo, en nuestra implementación, partimos de una señal de error definida como $(t - y)$, que ya era opuesta al gradiente del error ($y - t$), motivo por el que la actualización de los pesos se realiza incrementándolos proporcionalmente a los deltas calculados. Se trata de una estrategia habitual en muchas implementaciones de *backpropagation*. Si hubiésemos utilizado el gradiente real del error, $(y - t)$, la actualización de los pesos se tendría que haber realizado en sentido opuesto, tal como indica el método del gradiente descendente: $\Delta w = -\eta \nabla_w E$.

La implementación que hemos realizado del método de ajuste de los parámetros de una red neuronal multicapa es una implementación básica,

Esta forma de representar la señal de error es habitual en sistemas de control, en los que la señal $(t - y)$ representa la diferencia entre lo que se desea obtener como salida y la salida que se obtiene realmente. Esta diferencia se envía, como realimentación, al sistema de control que se encarga de ajustar los parámetros del proceso que da lugar a la salida observada.

que podríamos mejorar de múltiples formas. Aunque más adelante analizaremos con detalle múltiples aspectos relacionados con el entrenamiento de redes neuronales, como anticipo podemos mencionar algunas mejoras que resultan casi inmediatas:

- Para acelerar la ejecución del algoritmo de propagación de errores, si utilizamos funciones de activación cuya derivada $f'(z)$ se pueda expresar en términos de la propia función $f(z)$, como sucede con la función logística y la tangente hiperbólica, podemos aprovecharnos de que la función $f(z)$ ya la hemos evaluado en la propagación hacia delante de la señal de entrada a través de la red. Si conservamos el valor de $f(z)$, podemos evaluar su derivada cuando hacemos la propagación hacia atrás de la señal de error sin tener que realizar el cálculo de funciones trascendentales.

Por ejemplo, si tenemos el valor de `Sigmoid.eval(z)` en la variable `y`, podemos calcular su derivada utilizando directamente la expresión `y*(1-y)`, ahorrándonos la llamada a `Sigmoid.diff(z)`, la cual involucra una llamada a la función trascendental `exp(-z)`. De esa forma, conseguimos agilizar el proceso de propagación de la señal de error a través de la red.

Éste es uno de los motivos por los que se suelen escoger funciones de activación que resulten fáciles de derivar (p.ej. ReLU) o que se puedan expresar en términos de la propia función, ya evaluada.

- Experimentalmente, se ha comprobado que existen variantes de *back-propagation* que reducen el número de iteraciones necesarias para que el algoritmo converja. Cada iteración del algoritmo de propagación de errores involucra dos pasadas a través de la red, una hacia adelante y otra hacia atrás, que pueden resultar costosas computacionalmente cuando las redes neuronales son grandes. De forma que cualquier modificación del algoritmo que nos ayude a reducir el número de iteraciones necesarias será siempre bienvenida.

Por ejemplo, Tariq Samad, de Honeywell, observó que el algoritmo de propagación de errores converge más rápidamente si se utilizan los valores esperados de las unidades de la red para actualizar sus pesos.¹⁸² Esos valores esperados se pueden aproximar como la suma de la salida de la neurona y el término de error que se emplea habitualmente en las actualizaciones de los pesos.

En vez de utilizar la expresión habitual para actualizar los pesos,

$$\Delta w_{ij}^k = -\eta x_i \delta_j^k$$

se suelen obtener mejores resultados, en lo que respecta a la convergencia del algoritmo, si utilizamos

$$\Delta w_{ij}^k = -\eta(x_i + \beta \delta_j^k) \delta_j^k$$

¹⁸² Tariq Samad. Back propagation with expected source values. *Neural Networks*, 4(5):615 – 618, 1991. ISSN 0893-6080. doi: 10.1016/0893-6080(91)90015-W

donde β es una constante (p.ej. $\beta = 1$).

- Otra posibilidad consiste en utilizar momentos en las actualizaciones de los pesos, partiendo de una analogía física. Imaginemos un trineo deslizándose por una colina nevada. Su inercia le hará superar pequeños altibajos (mínimos locales) en su camino descendente hasta el fondo del valle (mínimo global). Al utilizar el gradiente descendente, corremos el riesgo de quedarnos atrapados en un mínimo local de la superficie de error y no pasar de ahí. Si a nuestro movimiento le añadimos cierto grado de inercia, tal vez seamos capaces de hacer que el algoritmo funcione como el trineo que se desliza hasta el fondo del valle.

Existen múltiples variaciones de la estrategia basada en momentos. En su versión estándar clásica, utilizada ya en el artículo original de 1986,¹⁸³ la actualización de los pesos se efectúa de la siguiente forma:

$$\Delta w_{ij}^k(t) = -\eta x_i \delta_j^k + \alpha \Delta w_{ij}^k(t-1)$$

donde α es una constante que se denomina momento.

Si nos fijamos en la expresión anterior, lo único que hemos hecho es añadir un nuevo término a la fórmula de actualización de los pesos. Este término depende de un coeficiente (el momento α) y de la dirección en la que hicimos el último cambio en el vector de pesos (el de la iteración anterior). Obviamente, lo normal es que asumamos que $\Delta w_{ij}^k(0) = 0$.

La denominación “momento” no resulta demasiado acertada desde el punto de vista físico, aunque se mantiene por cuestiones históricas. En realidad, el valor $(1 - \alpha)$ se puede interpretar como un coeficiente de fricción en el movimiento del vector de pesos. Cuando $\alpha \rightarrow 0$, tenemos la fórmula típica del gradiente descendente, sin momentos: la fricción de nuestro movimiento con la superficie sobre la que nos movemos es absoluta, por lo que el movimiento se reduce al impulso que recibimos en cada momento a través de la señal asociada al gradiente del error. Cuando $\alpha = 1$, el movimiento se realiza sin fricción alguna, por lo que la única forma de detenernos es encontrarnos con un gradiente opuesto: nos pasamos del mínimo y oscilamos alrededor de él. Si $\alpha > 1$, tenemos un movimiento que acelera solo, sin intervención externa, por lo que habríamos inventado una máquina de movimiento perpetuo. Como es lógico, no es algo que nos sirva de mucho si lo que queremos es detenernos lo más cerca posible del mínimo.

Por tanto, lo normal es que utilicemos $0 \leq \alpha \leq 1$. El término asociado al momento, $\alpha \Delta w_{ij}^k(t-1)$, lo podemos interpretar entonces como una forma de mantener una media móvil de las contribuciones del gradiente descendente a los cambios en los pesos de la red. El uso de momentos suaviza los cambios en los pesos y, en general, suele acelerar el proceso de aprendizaje de la red. Encontrar un valor adecuado para los parámetros del algoritmo, la tasa de aprendizaje η y el momento α , suele consistir en un proceso de prueba y error.

¹⁸³ David E. Rumelhart, Geoffrey E. Hinton, y Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, October 1986a. doi: 10.1038/323533a0

El proceso de entrenamiento

Ya tenemos todos los ingredientes necesarios para poder entrenar una red neuronal multicapa. Dada una señal de entrada, podemos propagarla a través de la red para obtener una salida usando el método `forward`. Podemos calcular el gradiente del error utilizando una función auxiliar, `error`. Una vez que disponemos de este gradiente, podemos propagar el error hacia atrás con *backpropagation* mediante el uso de `backward`, que se encarga de calcular los deltas necesarios para el ajuste de los parámetros de la red. Por último, realizamos este ajuste mediante `update`, método que podemos modificar para agilizar el proceso de entrenamiento con momentos.

El proceso completo de entrenamiento de una red neuronal consiste en repetir múltiples veces el ciclo `forward-backward-update`. En el caso del aprendizaje *online*, realizamos una actualización de los pesos cada vez que le presentamos a la red un ejemplo del conjunto de entrenamiento. En el caso del aprendizaje por lotes [*batch learning*], primero le presentamos a la red todos los ejemplos del conjunto de entrenamiento, para los que vamos acumulando errores (el error de salida y los deltas asociados a cada neurona de la red) y, a continuación, realizamos una única actualización de los pesos. Obviamente, el entrenamiento *online* suele ser más eficiente, especialmente si utilizamos un ordenador que ejecute el algoritmo secuencialmente. El aprendizaje por lotes puede hacerse más rápido si aprovechamos que la evaluación de cada ejemplo del conjunto de entrenamiento puede realizarse en paralelo, ya que se utilizan los mismos pesos para todos los ejemplos.

Habitualmente, recorreremos varias veces el conjunto de datos de entrenamiento para ajustar los parámetros de la red neuronal. Cada uno de esos recorridos recibe el nombre de épocas. Así pues, podemos modelar el algoritmo de aprendizaje que combina *backpropagation* con gradiente descendente mediante un bucle anidado:

```
function train (net, set, eta)
    for epoch=1:EPOCHS
        for k=1:size(set)
            input = set.data[k];
            target = set.target[k];
            output = net.forward(input);
            error = error(target,output);
            net.backward(error);
            net.update(eta);
```

La función anterior entrena una red neuronal `net` a partir de un conjunto de entrenamiento `set` utilizando una tasa de aprendizaje `eta` y aprendizaje *online*. El aprendizaje *online* utiliza una estimación instantánea del gradiente del error, calculada a partir de un único ejemplo del

Con ayuda de una GPU de propósito general [GPGPU: General Purpose Graphics Processing Unit], que resulta especialmente indicada para realizar operaciones con matrices como las utilizadas en redes neuronales, se pueden conseguir mejoras significativas en la eficiencia del proceso con respecto al uso de un microprocesador convencional.

conjunto de entrenamiento. Esto hace que la estimación del gradiente no sea demasiado fiable e introduzca cierta aleatoriedad en el método del gradiente descendente, motivo por el que se conoce como gradiente descendente estocástico. El “ruido” introducido por la estimación estocástica del gradiente puede resultar incluso positivo en el proceso de optimización: puede ayudar a la hora de evitar mínimos locales de la función de error. En la práctica, esta estrategia de aprendizaje resulta más eficiente en lo que respecta a la convergencia del algoritmo.

A diferencia del aprendizaje *online*, el aprendizaje por lotes realiza una estimación más fiable del gradiente del error, ya que utiliza para ello el conjunto completo de datos de entrenamiento. Esto permite que la convergencia al mínimo (local) de la función de error sea más estable, al realizar cada actualización en la dirección correspondiente a la media de los gradientes obtenidos para cada ejemplo del lote.

Si queremos utilizar aprendizaje por lotes, realizaríamos una única actualización de los pesos (**update**) para cada época del entrenamiento, en vez de realizarla para cada ejemplo. Además, tendríamos que modificar ligeramente la implementación del método **backward** para que fuese acumulando los deltas y añadir un nuevo método que los inicializase (**reset**). El resultado tendría el siguiente aspecto:

```
function train (net, set, eta)
    for epoch=1:EPOCHS
        net.reset();
        for k=1:size(set)
            input = set.data[k];
            target = set.target[k];
            output = net.forward(input);
            error = error(target,output);
            net.backward(error);
        net.update(eta);
```

En las implementaciones anteriores hemos supuesto que el entrenamiento consiste en recorrer el conjunto de datos un número fijo de veces, dado por la constante **EPOCHS**. En vez de utilizar un número predeterminado de épocas de entrenamiento, se pueden emplear otros criterios para detener el proceso de entrenamiento de una red neuronal:

- Podemos evaluar el error cometido en cada época del aprendizaje y detener el entrenamiento cuando disminuya por debajo de un error que consideremos aceptable o llevemos varias épocas consecutivas en las que el error ha dejado de disminuir. En cualquier caso, siempre debemos mantener un criterio de parada estricto (p.ej. un número de épocas) ya que, si no elegimos adecuadamente parámetros del algoritmo como la tasa de aprendizaje, puede que el error evaluado

El gradiente descendente estocástico también será el método que utilicemos para entrenar una red que recibe sus datos de entrada provenientes de flujos continuos de datos, habitualmente conocidos como *data streams* (p.ej. datos de sensores en tiempo real).

En *deep learning* existe una tercera alternativa, a medio camino entre el aprendizaje *online* y el aprendizaje por lotes, conocida como aprendizaje por mini-lotes. En el aprendizaje por mini-lotes, se escoge aleatoriamente una muestra del conjunto de datos y se utiliza una estimación estocástica del gradiente a partir de las muestras seleccionadas. Generalmente, el tamaño de la muestra vendrá determinado por la capacidad de cálculo paralelo del hardware sobre el que ejecutemos el algoritmo.

sobre el conjunto de entrenamiento nunca disminuya. Por ejemplo, si la tasa de aprendizaje es demasiado elevada, el gradiente descendente puede que, no sólo no converja, sino que sea inestable y diverja.

- Otra posibilidad, muy habitual en la práctica, es emplear un conjunto de datos de validación, independiente tanto del conjunto de entrenamiento como del conjunto de prueba. Este conjunto de validación lo podemos emplear al final de cada época del aprendizaje para evaluar cómo se comporta la red neuronal al utilizarla sobre datos diferentes a los del conjunto de entrenamiento. Cuando observemos que el error sobre este conjunto de validación comienza a subir, es posible que estemos comenzando a sobreaprender [*overfitting*] y resulta aconsejable detener el entrenamiento de la red. Este criterio de parada se conoce con el término *early stopping*, que podríamos traducir como parada temprana.

Utilicemos la estrategia que utilicemos, algo que debemos tener en cuenta es que los métodos de optimización como el gradiente descendente, en principio, pueden quedar atrapados en un óptimo local de la función de error. En tal caso, puede que resulte aconsejable comenzar de nuevo el proceso de entrenamiento desde una configuración inicial diferente de los parámetros de la red, algo de lo que nos ocuparemos a continuación.

Inicialización de los pesos

Es importante inicializar correctamente los parámetros de las red antes de comenzar el proceso de entrenamiento: los pesos de las sinapsis y los sesgos de las neuronas.

Un error de principiante muy habitual consiste en inicializar todos los pesos a cero. Sin embargo, no es una buena idea. Aparte de que puede corresponder a un mínimo local de la función de error, impide que la red neuronal pueda aprender correctamente. En general, siempre que utilicemos los mismos valores para inicializar los pesos de las distintas neuronas de la red, el algoritmo de aprendizaje no funcionará. ¿Por qué? Porque el gradiente del error será siempre el mismo para todas las neuronas, la corrección realizada a partir del gradiente será la misma... y todas las neuronas tendrán siempre los mismos pesos. En definitiva, estaremos entrenando réplicas de neuronas individuales, un perceptrón en vez de una red neuronal.

Lo más habitual consiste en inicializar los pesos de la red de forma aleatoria. Si disponemos de un generador de números pseudoaleatorios `random` que genere valores aleatorios de acuerdo a una distribución uniforme en el intervalo $[0, 1]$, podemos inicializar cada peso de la red utilizando una expresión como:

```
w = 2.0 * (random() - 0.5) * SCALE;
```

donde `(random()-0.5)` proporciona valores aleatorios en el intervalo $[-0.5, 0.5]$, que multiplicamos por 2 para que correspondan al intervalo $[-1, 1]$ y por una constante `SCALE` mediante la que determinamos la escala que deseamos que tengan nuestros pesos (en este caso, su valor máximo). Este último valor debe escogerse de tal forma que, si utilizamos funciones sigmoidales como funciones de activación de nuestras neuronas, las sigmoides nunca se saturen antes de comenzar el entrenamiento, algo que ya sabemos que puede ralentizar el proceso de entrenamiento.

Observe, además, que la inicialización de los pesos la hemos realizado para que haya tanto pesos iniciales positivos como pesos iniciales negativos. En una red neuronal, especialmente si utilizamos neuronas con niveles de activación en el intervalo $[0, 1]$, resulta de interés que existan tanto sinapsis excitatorias (pesos positivos) como sinapsis inhibitorias (pesos negativos).

Aunque no lo hemos mencionado, la inicialización de los sesgos de las neuronas puede realizarse de la misma forma que la de los pesos o, incluso, podemos inicializarlos a 0 y dejar que sea el proceso de entrenamiento de la red el que determine sus valores más adecuados.

Preprocesamiento de los datos de entrada

Aunque el algoritmo de entrenamiento de una red neuronal no requiere que preprocesemos los datos de entrada, puede que determinadas transformaciones de los datos de entrada nos ayuden a mejorar el rendimiento del algoritmo de aprendizaje. Podemos normalizar los vectores de entrada, por ejemplo, realizando una normalización sobre el intervalo $[0, 1]$ para cada una de las variables de nuestros ejemplos de entrenamiento:

$$x_{[0,1]} = \frac{x}{\max - \min}$$

Otra alternativa muy utilizada en minería de datos consiste en el uso de *z-scores*, que normalizan los datos con respecto a su media y desviación:

$$z = \frac{x - \mu}{\sigma}$$

donde μ representa la media y σ la desviación estándar. Esta transformación consigue que la variable estandarizada x_z tenga media 0 y desviación típica 1. De forma que el *z-score* z asociado a un valor x indica el número de desviaciones típicas que el valor se encuentra por encima o debajo de la media, motivo por el que también se conoce a z como unidad tipificada.

Centrar una variable consiste en restarle su media a cada uno de sus valores y reducir una variable consiste en dividir sus valores por su desviación típica, por lo que la variable estandarizada también se conoce por el nombre de variable centrada reducida.

Una función sigmoidal saturada da lugar a una derivada prácticamente nula. Como esta derivada interviene en el cálculo del gradiente del error, estaremos impidiendo que la red sea capaz de aprender correctamente los pesos.

En análisis de datos, también es habitual denominar al *z-score* usando el término variable normalizada, lo que resulta demasiado ambiguo. Nosotros reservaremos el término normalización para la normalización en el intervalo [0,1] y hablaremos de *z-scores* para las unidades tipificadas.

Presentación de los datos de entrada

Cuando entrenamos una red neuronal, si siempre le presentamos los datos de entrenamiento en el mismo orden, es posible que comiencen a aparecer oscilaciones en los pesos de la red. Por este motivo, resulta aconsejable que, en cada época de entrenamiento, le mostremos a la red los datos en un orden diferente. Para conseguirlo, basta con que barajemos los datos al comienzo de cada época, por lo que añadiríamos una nueva línea al algoritmo de entrenamiento de la red:

```
function train (net, set, eta)
    for epoch=1:EPOCHS
        set.shuffle()
        ...
    end
```

La llamada al método `shuffle` se encarga de reordenar los datos del conjunto de entrenamiento antes de comenzar una época. La implementación de este método se puede añadir a nuestra clase `Dataset` de la siguiente forma:

```
class Dataset:
    function shuffle()
        n = size()
        for i=1:n-1
            p = random(n-i+1);
            swap(i,i+p);

    function swap(i,j)
        tmp = (data[i],target[i]);
        (data[i],target[i]) = (data[j],target[j]);
        (data[j],target[j]) = tmp;
```

donde `random(n)` genera un número aleatorio uniforme entre 1 y *n*, que podemos implementar a partir de un generador aleatorio uniforme como `1+n*random()`, mientras que `swap(i,j)` intercambia los patrones *i* y *j* del conjunto de datos de entrenamiento.

En nuestra implementación de `shuffle` hemos sido cuidadosos para que todas las permutaciones del conjunto de datos de entrada sean equiprobables. Para elegir el primer elemento de la permutación utilizamos un número aleatorio entre 1 y *n*, siendo *n* el tamaño del conjunto de datos, de forma que todos los ejemplos del conjunto de entrenamiento

tienen una probabilidad $1/n$ de ser seleccionados en primer lugar. Para elegir el segundo elemento, utilizamos un número aleatorio entre 1 y $n - 1$. De esta forma, ya elegido el primer elemento de la permutación, elegimos entre los $n - 1$ elementos que no fueron elegidos en primera opción. La probabilidad de que un elemento termine en segunda posición es, por tanto, la probabilidad de que no sea elegido el primero $(n - 1)/n$ por la probabilidad de ser escogido el segundo $1/(n - 1)$: $p = (n - 1)/n * 1/(n - 1) = 1/n$. El tercer elemento lo elegiremos entre los $n - 2$ restantes y así sucesivamente, hasta llegar al penúltimo, que elegiremos de entre los dos últimos con probabilidad $1/2$. Es la forma de garantizar que todas las permutaciones posibles del conjunto de datos de entrenamiento se eligen con la misma probabilidad.

Hiperparámetros del algoritmo

El hiperparámetro más importante de nuestro algoritmo de entrenamiento de redes neuronales multicapa es su tasa de aprendizaje η (**eta**). Si se fija un valor excesivamente bajo, los ajustes de los pesos se irán haciendo muy poco a poco, por lo que el entrenamiento de la red será innecesariamente lento y requerirá muchas iteraciones. Si fijamos un valor excesivamente alto, los cambios en los pesos pueden ser demasiado bruscos y los pesos pueden oscilar de forma salvaje, lo que puede entorpecer el aprendizaje e incluso prevenirllo por completo.

Así pues, es necesario encontrar un valor adecuado para la tasa de aprendizaje, ni demasiado bajo, ni demasiado elevado. Por desgracia, el valor más adecuado dependerá del problema que pretendamos resolver y de la topología de nuestra red, por lo que no existen criterios objetivos para establecer un valor que funcione bien. Usualmente, se comienza utilizando un valor pequeño, siempre menor que 1: `eta=0.1`, por ejemplo. A continuación, en función de lo que observemos al entrenar la red, podemos ajustar la tasa de aprendizaje: duplicarla si vemos que el aprendizaje va demasiado lento o dividirla por la mitad si el algoritmo diverge. Una vez que encontramos la escala adecuada para la tasa de aprendizaje, podemos intentar afinarla si disponemos de tiempo para ello.

Si además utilizamos momentos, también tendremos que encontrar un valor adecuado para el momento α . Por desgracia, los distintos hiperparámetros del algoritmo interactúan entre sí, por lo que no se pueden establecer de forma independiente. Esto es, el valor más adecuado para la tasa de aprendizaje η depende, no solamente del problema y de la topología de la red, sino de qué valor hayamos escogido para el momento α . Este tipo de interacciones hace que el correcto entrenamiento de una red neuronal siga siendo más un arte que una ciencia. Un arte en el que la experiencia y perspicacia del científico de datos pueden resultar cruciales para aprovechar al máximo las capacidades ofrecidas por las

redes neuronales artificiales.

Depuración de la implementación

La implementación de algoritmos de minería de datos, especialmente si son de tipo estocástico, ha de realizarse con extremo cuidado. Entre otros motivos, porque puede que, aun realizando una implementación incorrecta de un algoritmo de aprendizaje, obtengamos resultados que parecen razonables (aunque pueden no ser todo lo buenos que deberían).

En general, a la hora de realizar la implementación de cualquier tipo de algoritmo, es aconsejable diseñar baterías de casos de prueba. Los casos de prueba sirven, en primer lugar, para comprobar que el código que hemos escrito funciona correctamente. Es decir, hace lo que tiene que hacer, generando los resultados que de él esperamos. Esos casos de prueba los podemos automatizar con la ayuda de herramientas como JUnit, en el caso de Java, <http://junit.org/>. La automatización de la ejecución de los casos de prueba nos ayuda, además, a comprobar que las modificaciones que realizamos sobre el código no introducen nuevos errores, que podrían pasar inadvertidos sin la ayuda de herramientas automatizadas. Esto es, la automatización nos permite realizar automáticamente tests de regresión. Como ventaja añadida, reducimos el tiempo que tardamos en depurar errores, ya que si ejecutamos con frecuencia los casos de prueba nos daremos cuenta del error justo cuando lo acabamos de cometer (p.ej. antes de hacer cada *commit* en nuestra herramienta de control de versiones).

En el caso de las redes neuronales artificiales, hemos de ser especialmente cuidadosos. A la hora de depurar la implementación de un algoritmo de entrenamiento de redes neuronales, podemos utilizar diferentes estrategias:

- Visualizar el sistema en funcionamiento: Si nos fijamos únicamente en las señales de error, tal vez nos estemos perdiendo algo. Imagine lo que podría pasar si el error está en la rutina encargada de la evaluación del error...
- Visualizar los errores de la red: Nos puede ayudar a detectar problemas, no sólo en el algoritmo de aprendizaje, sino en el preprocesamiento de los datos.
- Razonar acerca de los errores sobre los conjuntos de entrenamiento y prueba: Si el error sobre el conjunto de entrenamiento es bajo pero se mantiene muy elevado sobre el conjunto de prueba, tal vez no estemos aprendiendo correctamente. O tal vez no estemos evaluando correctamente el error sobre el conjunto de prueba (puede que no preprocesemos los datos de prueba igual que los de entrenamiento, por lo que la red nunca podrá funcionar correctamente).

Hay quien incluso defiende que, antes de comenzar a escribir código, se deben implementar los tests que nos permitirán comprobar que el código funciona correctamente. Es el desarrollo dirigido por pruebas o TDD [*Test-Driven Development*]. Aunque a primera vista pueda parecer algo extraño, de esta forma nos podemos centrar más fácilmente en qué necesitamos hacer para que nuestro software funcione, hacemos hincapié en la interfaz de nuestro sistema antes que en su implementación (algo siempre positivo desde el punto de vista del diseño de software) y definimos claramente cuándo termina nuestro trabajo (cuando se pasan con éxito todos los tests).

- Monitorizar el algoritmo de entrenamiento en sí: La simple visualización de los niveles de activación de las neuronas y de los gradientes del error puede ayudarnos a detectar problemas relacionados con la saturación de las neuronas, la desaparición o la explosión del gradiente. Así mismo, deberíamos comprobar cuál es la magnitud de los gradientes con respecto a los parámetros que ajustan: las actualizaciones realizadas sobre los parámetros deberían ser moderadas en un algoritmo como el gradiente descendente (p.ej. el 1 % del valor, no el 0.01 % ni el 200 %).

Como el proceso de depuración de un algoritmo de este tipo involucra ejecutarlo múltiples veces, resulta más que aconsejable realizar pruebas con conjuntos de datos pequeños, o bien muestras de los conjuntos de datos reales sobre los que trabajaremos en la práctica o bien diseñados específicamente para la realización de pruebas.

Si conocemos algunos invariantes que nuestro algoritmo ha de cumplir (condición o propiedades necesariamente ciertas en ciertos puntos de la ejecución de nuestro algoritmo), puede ser recomendable que instrumentemos nuestro código para que se compruebe automáticamente la validez de esos invariantes.

Sobre el cálculo del gradiente

El aspecto clave del entrenamiento de redes neuronales usando *backpropagation* y gradiente descendente es, precisamente, el cálculo del gradiente del error. Es fundamental que nos aseguremos de que, numéricamente, estamos calculando correctamente las derivadas parciales involucradas en la propagación del error hacia atrás.

Para realizar esta comprobación, que idealmente incluiremos en nuestra batería de pruebas automatizadas, podemos recordar la definición de la derivada de una función que aprendimos en el instituto:

$$\frac{dJ(\theta)}{d\theta} = \lim_{\epsilon \rightarrow 0} \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon}$$

donde J es la función que estamos derivando y θ es su conjunto de parámetros.

En vez de utilizar esta definición de la derivada, emplearemos una diferencia centrada (en torno a θ) para mejorar la precisión de la aproximación numérica de la derivada de una función por medio de una diferencia finita:

$$\frac{dJ(\theta)}{d\theta} = \lim_{\epsilon \rightarrow 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

Para cualquier valor de θ , podemos aproximar numéricamente la derivada de la función $J(\theta)$ usando un valor pequeño de ϵ (epsilon). Un valor demasiado pequeño podría ocasionar errores de redondeo y

Aunque en el texto hemos utilizado E y w para hacer referencia a la función de error y sus parámetros, los pesos de la red, en aprendizaje automático es habitual utilizar la notación $J(\theta)$ para hacer referencia a las funciones de coste, pérdida o error.

underflow, por lo que no conviene apurar demasiado en este sentido. Por ejemplo, nos basta con establecer $\epsilon = 10^{-4}$ y usar la siguiente expresión:

$$g(\theta) = \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon} \approx \frac{dJ(\theta)}{d\theta}$$

Normalmente, no trabajaremos con derivadas de funciones de una sola variable, sino con gradientes (vectores de derivadas parciales de una función escalar con respecto a un vector de variables) y matrices jacobianas (derivadas parciales de funciones vectoriales con respecto a vectores de variables). Así pues, dado que θ es un vector de parámetros, no un simple número real, tendremos que comprobar que el gradiente se calcula correctamente para cada variable individual θ_i .

Para cada vector de parámetros θ , evaluaremos $g_i(\theta)$ como una aproximación de $\partial J(\theta)/\partial\theta_i$. Para ello, definimos $\theta_{i+} = \theta + \epsilon\vec{e}_i$, donde \vec{e}_i es un vector *one-hot*, con un uno en su posición i-ésima y ceros en las demás:

$$\vec{e}_i = \begin{bmatrix} 0 & 0 & \dots & 1 & \dots & 0 \end{bmatrix}^\top$$

θ_{i+} es similar a θ , salvo que su i-ésima componente se ve incrementada por ϵ . De la misma forma, podemos obtener $\theta_{i-} = \theta - \epsilon\vec{e}_i$, el vector θ con su i-ésima componente reducida en ϵ . Estos vectores nos servirán para verificar numéricamente el gradiente con respecto a cada parámetro:

$$g_i(\theta) = \frac{J(\theta_{i+}) - J(\theta_{i-})}{2\epsilon}$$

Para depurar nuestra implementación del cálculo de gradientes sólo tenemos que repetir el cálculo para diferentes valores de θ , con el objetivo de comprobar que la diferencia entre los valores calculados por nuestra implementación de un gradiente y los valores aproximados numéricamente no difieren en exceso.

Implementación modular

Aunque la implementación que hemos realizado del algoritmo de propagación de errores, *backpropagation*, combinado con un método de optimización basado en el gradiente descendente es correcta y puede resultarnos útil para resolver multitud de problemas, en *deep learning* es habitual realizar experimentos con la topología de la red y combinar distintos tipos de módulos para construir redes neuronales complejas. Puede que nos interese utilizar distintas funciones de activación para distintas capas de la red, jugar con diferentes formas de conectar una capa con la siguiente, permitir conexiones entre capas no adyacentes [*skip connections*] o, incluso, crear ciclos para formar redes recurrentes. El principal atractivo del *deep learning* (y puede que su maldición) es que podemos ajustar montones de hiperparámetros a la hora de diseñar la arquitectura de una red neuronal, que construiremos como si de una maqueta de LEGO se tratase.

Para construir redes complejas combinando distintos tipos de módulos, resulta aconsejable realizar una implementación más modular del algoritmo de propagación de errores. En un diseño modular, cada módulo de los que podremos utilizar para construir una red neuronal se encargará de realizar dos operaciones:

- Propagación hacia adelante: Calcular una salida y a partir de una entrada dada x de acuerdo a sus parámetros w (o θ , si lo prefiere).
- Propagación hacia atrás: Propagar una señal de error $\nabla_y E$, desde la salida del módulo hacia atrás. No sólo hacia sus parámetros $\nabla_w E$, sino también hacia sus entradas $\nabla_x E$.

La señal de error asociada a las entradas $\nabla_x E$ se la podemos suministrar como señal de error a la salida de los módulos de donde provienen esas entradas x . Es decir, si tenemos un módulo A cuya salida y_A se utiliza como entrada x_B en un módulo B : $\nabla_{y_A} E = \nabla_{x_B} E$. De esta forma, podemos construir configuraciones arbitrarias de módulos y calcular todos los gradientes ∇E que podamos necesitar.

Volviendo al ejemplo concreto de la red neuronal multicapa, veamos cómo podemos aplicar este diseño modular a nuestra implementación. En primer lugar, creamos una clase abstracta para representar las capas de una red neuronal: la clase `Layer`. Esta clase nos servirá de base en el futuro para crear distintos tipos de redes neuronales. En principio, sólo necesitamos que realice dos operaciones:

```
class Layer(n,m):
    function y = forward(x);
    function backward(error);
```

donde `n` indica el número de entradas y `m` el número de salidas de la capa.

Hasta ahora, el único tipo de red neuronal que hemos utilizado es el formado por múltiples capas completamente conectadas, en el que todas las salidas de una capa se utilizan como entradas en todas las neuronas de la capa siguiente de la red. Así que definimos una clase `FullyConnectedLayer` que encapsule la funcionalidad de este tipo de capas.

El método `forward` es el encargado de propagar una señal de entrada `x` para obtener una salida `y`:

```
class FullyConnectedLayer(n,m): Layer(n,m)
    function y = forward(x)
        for j=1:m
            z = b[j]
            for i=1:n
                z += w[j][i]*x[i];
            y[j] = activation.eval(z);
```

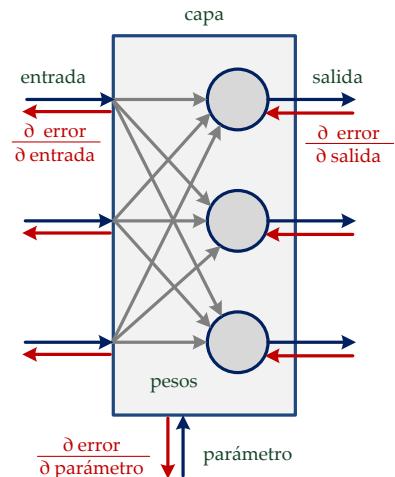


Figura 110: Diseño modular de los componentes de una red neuronal artificial.

Por ahora, las entradas y salidas de la red vienen en forma de vectores, por lo que `n` y `m` serán números enteros. Cuando trabajemos con imágenes, las entradas y salidas pueden ser matrices (arrays bidimensionales). En general, pueden ser tensores (arrays multidimensionales), cuyas dimensiones vendrán dadas por `n` y `m`.

El método `backward` será el encargado de propagar una señal de error hacia atrás y generar todos los “deltas” necesarios. En este caso, tenemos que calcular el gradiente del error con respecto a los pesos `w` (`deltaW`), con respecto a los sesgos `b` (`deltaB`) y también con respecto a las entradas `x` (`deltaX`). Los tres gradientes los calculamos a partir del gradiente con respecto a la entrada neta de la neurona, nuestro antiguo `delta`. La implementación del método queda como se muestra a continuación:

```
class FullyConnectedLayer(n,m): Layer(n,m)
    function backward(error)
        for j=1:m
            // dE/dz
            delta[j] = error[j] * activation.diff(z[j]);
            // dE/db (con respecto al sesgo o bias)
            deltaB[j] = delta[j];
            // dE/dw (con respecto a los pesos w)
            for i=1:n
                deltaW[j][i] = delta[j] * x[i];
            // dE/dx (con respecto a las entradas x)
            for i=1:n
                deltaX[i] = 0;
                for j=1:m
                    deltaX[i] += delta[j] * w[j][i];
```

Una vez que hemos encapsulado la funcionalidad completa de una capa de la red multicapa, la implementación en sí de la red completa se simplifica. Sigue estando formada por un array de capas, pero su implementación resulta más simétrica que en la versión que vimos anteriormente. Aquí tenemos el resultado:

```
class FeedForwardNetwork(n,m): Layer(n,m)

    function y = forward(x)
        y = x;
        for k=1:size(layer)
            y = layer[k].forward(y);

    function backward(error)
        for k=size(layer):1
            layer[k].backward(error);
            error = layer[k].deltaX;
```

Como beneficio extra, podemos observar que la red multicapa, en su conjunto, tiene la misma interfaz que cada una de las capas que la componen, por lo que la podemos utilizar como un módulo más con el que componer otras redes más complejas. Aquí reside la belleza del diseño modular en las herramientas de *deep learning*.

Implementación en una GPU

Dado que las operaciones realizadas en la propagación de señales a través de redes neuronales artificiales son, básicamente, operaciones con matrices, su implementación se puede beneficiar de la potencia de cálculo que ofrecen las GPU [*Graphical Processing Units*]. Originalmente diseñadas para realizar de forma eficiente las operaciones geométricas típicas que requieren aplicaciones gráficas como los videojuegos, hoy en día ofrecen su capacidad de cálculo para realizar otras tareas que puedan aprovechar su arquitectura paralela. Es el caso de las técnicas de *deep learning*, cuyas operaciones se pueden representar en notación vectorial, lo que simplifica su paralelización.

Para programar una GPU de propósito general existen distintos API [*Application Programming Interface*] estándar. El más conocido de ellos es, sin duda, CUDA. El estándar CUDA, que originalmente significaba *Compute Unified Device Architecture*, es el utilizado por las GPU Nvidia, si bien existen otros estándares que pretenden ser independientes del fabricante de hardware, como es el caso de DirectCompute de Microsoft (parte de DirectX en Windows) y de OpenCL [*Open Computing Language*], promovido por el consorcio tecnológico Khronos Group (más conocido por OpenGL, muy utilizado en informática gráfica).

Utilizando CUDA, la implementación eficiente de una capa se puede realizar de la siguiente manera:

```
class CUDAFullyConnectedLayer(n,m): FullyConnectedLayer(n,m)

    function y = forward(x)
        cublasSgemm(1,w,x,0,z); // z = wx
        cublasSgemm(1,b,1,1,z); // z += b
        cudnnActivationForward(activation,1,z,0,y);

    function backward(error)
        cudnnActivationBackward(activation,1,y,error,0,delta);
        // deltaW = x * delta
        cublasSgemm(1,x,delta,0,deltaW);
        deltaB = delta;
        // deltaX = w * delta
        cublasSgemm(1,w,delta,0,deltaX);
```

donde

- La operación `cublasSgemm` es la implementación, en la biblioteca Cu-BLAS de Nvidia, de la función *SGEMM* incluida en la especificación BLAS [Basic Linear Algebra Subprograms]. BLAS es un estándar de facto que define rutinas de cálculo matricial especialmente diseñadas

para su ejecución en arquitecturas paralelas de tipo SIMD [*Single Instruction, Multiple Data*], como es el caso de las GPU. En particular, la función $SGEMM(\alpha, A, B, \beta, C)$ realiza la siguiente multiplicación de matrices utilizando números en coma flotante de precisión simple (32 bits): $C = \alpha * A * B + \beta * C$.

- Las funciones `cudnnActivationForward` y `cudnnActivationBackward` son las funciones que Nvidia proporciona para realizar en paralelo la evaluación de la función de activación (`forward`) y su derivada (`backward`), tal como se hace cuando utilizamos *backpropagation*. Ambas están incluidas en la biblioteca CuDNN [*CUDA Deep Neural Networks*], ofrecida por Nvidia para ejecutar algoritmos de *deep learning* usando sus GPU.

Al lector más avisado tal vez le haya llamado la atención que utilicemos funciones para multiplicar dos matrices cuando una multiplicación de una matriz por un vector podría resultar más eficiente. Lo hemos hecho así porque, de esta forma, podemos aprovechar la capacidad de cálculo paralelo de una GPU para procesar no un ejemplo, sino un lote completo del conjunto de datos en una sola llamada a los métodos `forward` y `backward`. Esto permite acelerar el entrenamiento por lotes de una red neuronal (y también su uso en la práctica, una vez que la tengamos entrenada).

Variaciones sobre el tema

En función del tipo de problema que queramos resolver, introduciremos algunas variaciones sobre el algoritmo de aprendizaje de redes neuronales multicapa que hemos analizado en este capítulo. Basten un par de ejemplos:

- En problemas de regresión, nos interesa obtener una salida que no esté artificialmente acotada por el rango de la función de activación de las neuronas de la capa de salida, por lo que podemos utilizar unidades lineales en esta última capa de la red. La salida de estas neuronas será, simplemente, $y = w \cdot x$. El resto del algoritmo de entrenamiento se mantiene tal como lo hemos descrito.
- En problemas de clasificación, la señal de error adecuada no es la derivada del error cuadrático SSE, sino la entropía cruzada, que se calcula de la siguiente forma:

```
function error = crossEntropy (target, output)
    for i=1:size(output)
        error[i] = - target[i] * log(output[i])
        - (1.0-target[i]) * log(1.0-output[i]);
```

SGEMM [*Single-precision GGeneral Matrix Multiplication*] también está disponible para OpenCL en la biblioteca clBLAS, optimizada para GPUs de AMD, y en la biblioteca Intel MKL [*Math Kernel Library*] para procesadores Intel, incluyendo los Intel Xeon Phi de arquitectura MIC [*Many Integrated Core*] que llegan, como las GPU, a los teraflops: billones de operaciones en coma flotante por segundo (1 TFLOPS = 10^{12} FLOPS).

Cuando el problema de clasificación es binario, podemos utilizar una red con una sola neurona de salida, de tipo sigmoidal. Si tenemos un problema de clasificación con k clases diferentes, lo normal es que utilicemos una capa de tipo *softmax*, cuyas neuronas utilizan la siguiente función de activación:

$$y_j = f_{\text{softmax}}(z_j) = \frac{e^{z_j}}{\sum_i e^{z_i}}$$

y cuya derivada coincide con la derivada de la función logística:

$$\frac{\partial y_j}{\partial z_j} = y_j(1 - y_j)$$

Coda: Un poco de historia

El término *backpropagation*, estrictamente, corresponde únicamente a la fase en la que se propaga una señal de error hacia atrás para calcular eficientemente el gradiente del error con respecto a los diferentes parámetros de una red neuronal multicapa. Sin embargo, habitualmente se utiliza para hacer referencia al proceso completo de entrenamiento de una red, que incluye el cálculo del gradiente y su aplicación en un método de optimización basado en el gradiente descendente. En realidad, se puede utilizar en combinación con cualquier algoritmo de optimización basado en el gradiente, como el método de los gradientes conjugados o el algoritmo L-BFGS [*Limited-memory BFGS*], dos de los más utilizados en *deep learning*.

- La propagación del gradiente en *backpropagation* se basa en aplicar en repetidas ocasiones la regla de la cadena, representada usando la notación de Leibniz como

$$\frac{df}{dx} = \frac{df}{dg} \frac{dg}{dx}$$

Esta regla se conoce desde el siglo XVII. El matemático y filósofo alemán Gottfried Wilhelm (von) Leibniz, creador del cálculo diferencial, la mencionó en 1676 y se utiliza de forma implícita en el primer libro de texto de cálculo infinitesimal, publicado por el matemático francés Guillaume François Antoine, marqués de L'Hôpital, en 1696: *Analyse des Infiniment Petits pour l'Intelligence des Lignes Courbes* [análisis de lo infinitamente pequeño para comprender las líneas curvas]. En ese libro aparece por primera vez la regla de L'Hôpital, que en realidad fue demostrada por el matemático suizo Johann Bernoulli en 1694. La primera versión “moderna” de la regla de la cadena no apareció hasta un siglo después, en 1797, cuando el matemático italo-francés Joseph-Louis Lagrange publicó su tratado *Théorie des fonctions analytiques* [teoría de las funciones analíticas].

El método L-BFGS es una aproximación del algoritmo de Broyden–Fletcher–Goldfarb–Shanno [BFGS] que requiere una cantidad lineal de memoria, $O(n)$. A su vez, el algoritmo BFGS es una aproximación del método de Newton, que utiliza la matriz hessiana (las segundas derivadas parciales) para resolver problemas de optimización: el método de Newton requiere invertir esta matriz mientras que BFGS realiza una aproximación. Tanto el método de Newton como BFGS son cuadráticos en cuanto a consumo de memoria (la matriz hessiana requiere $O(n^2)$ espacio), lo que los hace inviables cuando trabajamos con redes neuronales de millones de parámetros.

La regla de L'Hôpital permite resolver límites que involucran indeterminaciones de la forma $0/0$ y ∞/∞ usando derivadas: $\lim_{x \rightarrow c} f(x)/g(x) = \lim_{x \rightarrow c} f'(x)/g'(x)$.

- El segundo ingrediente esencial de la receta que nos permite entrenar redes neuronales multicapa es el método de optimización del gradiente descendente. El método se le atribuye al barón Augustin-Louis Cauchy, un físico y matemático francés del siglo XIX, que el 18 de octubre de 1847 lo utilizó en un informe remitido a la Academia de las Ciencias de París: *Méthode générale pour la résolution de systèmes d'équations simultanées* [método general para la resolución de sistemas de ecuaciones simultáneas].

Así pues, todo lo que resulta realmente necesario para entrenar redes neuronales multicapa existe desde... mediados del siglo XIX. Visto así, no parece que el *deep learning* resulte tan novedoso.

De hecho, ni siquiera tendríamos que aplicar la regla de la cadena. Podríamos emplear método de optimización basado en el gradiente para ajustar los parámetros de una red neuronal cualquiera. Sólo tendríamos que estimar el gradiente del error E con respecto a un parámetro concreto de la red w_k realizando una aproximación numérica como la que empleamos para comprobar la corrección de nuestro cálculo del gradiente

$$\frac{\partial E}{\partial w_k} \approx \frac{E(w + \epsilon e_k) - E(w)}{\epsilon}$$

donde ϵ es un número positivo pequeño que utilizamos para introducir una perturbación en la red y ver cuál es la variación del error que se observa.

Es decir, podemos estimar el gradiente del error con respecto a cualquier parámetro $\partial E / \partial w_k$ evaluando la función de coste para dos valores ligeramente distintos del parámetro w_k . El algoritmo resultante es extremadamente sencillo, pero resultaría demasiado ineficiente. Si tenemos millones de pesos, tendríamos que evaluar $E(w + \epsilon e_k)$ millones de veces, lo que involucraría realizar millones de recorridos a través de la red neuronal.

La contribución de *backpropagation* es que podemos calcular todos esos gradientes realizando únicamente dos recorridos de la red: uno hacia adelante, para calcular su salida a partir de una entrada dada, y un segundo recorrido hacia atrás, para propagar la señal de error. Como los cálculos necesarios en ambos recorridos son similares, podemos decir que *backpropagation* reduce a sólo dos pasadas a través de la red lo que, de forma ingenua, podríamos conseguir realizando millones de recorridos.

El algoritmo de propagación de errores hacia atrás, *backpropagation*, ha sido redescubierto en varias ocasiones:

- *Teoría de control* (años 60)

En el fondo, *backpropagation* lo único que hace es aplicar la técnica de programación dinámica para evitar tener que recalcular una y otra vez los mismos gradientes. Por este motivo, no resulta demasiado

Como antes, e_k es un vector unitario en la dirección de la k -ésima dimensión, la correspondiente al parámetro w_k que deseamos ajustar.

sorprendente que el primer campo en el que se aplicase *backpropagation* fuese el de los sistemas de control, donde la programación dinámica es uno de sus caballos de batalla.

Henry J. Kelley, del Virginia Polytechnic Institute, lo utilizó en sistemas de control de trayectorias de vuelo para sistemas aeroespaciales en 1960.¹⁸⁴ Arthur E. Bryson, de la Universidad de Stanford, lo empleó para resolver problemas de optimización en 1961, en este caso de asignación de recursos.¹⁸⁵ Un año después, Stuart Dreyfus, que entonces era un estudiante de doctorado en Harvard y luego sería profesor de la Universidad de Berkeley, simplificó la notación empleada por Kelley y Bryson utilizando la regla de la cadena.¹⁸⁶

Arthur E. Bryson y Yu-Chi Ho describían en 1969 como un método de optimización de sistemas dinámicos multietapa, utilizando la terminología habitual de la programación dinámica.¹⁸⁷ En 1973, Stuart Dreyfus usaba *backpropagation* para adaptar los parámetros de un controlador en proporción a los gradientes de la señal de error.¹⁸⁸

■ *Diferenciación automática* (años 70)

In 1970, el matemático finlandés Seppo Linnainmaa publicó un método de diferenciación automática [*AD: Automatic Differentiation*] que permite calcular de forma eficiente la derivada de una función compuesta de funciones diferenciables y representada en forma de grafo.^{189,190} El método, de acumulación hacia atrás [*reverse accumulation*] o modo inverso [*reverse mode*], aplica recursivamente la regla de la cadena a los bloques que componen la función. ¿Le suena? Es la estrategia en que se basan las técnicas que automatizan el cálculo del gradiente si realizamos una implementación modular de una red neuronal compleja. En realidad, *backpropagation* aplicado a redes neuronales multicapa no es más que un caso especial de diferenciación automática en modo inverso, en el que tenemos una cadena lineal de bloques en vez de un grafo más complejo.

■ *Redes neuronales artificiales* (años 80)

En 1974, Paul Werbos mencionó, en su tesis doctoral,¹⁹¹ la posibilidad de aplicar lo que hoy conocemos como *backpropagation* al entrenamiento de redes neuronales. Unos años después, en 1982, utilizó el método de diferenciación automática de Linnainmaa en redes neuronales artificiales, tal como lo seguimos haciendo hoy.¹⁹²

En 1985, David Parker, del MIT, redescubrió una vez más el método de propagación de errores para ajustar los parámetros de una red neuronal.¹⁹³

Al año siguiente, en 1986, David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams publicaron su famoso artículo en la revista *Nature*. En ese artículo, mostraron que el ordenador era capaz de aprender extrayendo características de los datos que se representan en las capas

¹⁸⁴ Henry J. Kelley. Gradient Theory of Optimal Flight Paths. *ARS Journal*, 30(10):947–954, 1960. DOI: 10.2514/8.5282

¹⁸⁵ Arthur Earl Bryson. A gradient method for optimizing multi-stage allocation processes. En *Proceedings of the Harvard University Symposium on Digital Computers and Their Applications*, April 1961

¹⁸⁶ Stuart E. Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, 1962. DOI: 10.1016/0022-247X(62)90004-5

¹⁸⁷ Arthur Earl Bryson y Yu-Chi Ho. *Applied Optimal Control: Optimization, Estimation, and Control*. Blaisdell Publishing Company or Xerox College Publishing, 1969

¹⁸⁸ Stuart E. Dreyfus. The computational solution of optimal control problems with time lag. *IEEE Transactions on Automatic Control*, 18(4):383–385, Aug 1973. ISSN 0018-9286. DOI: 10.1109/TAC.1973.1100330

¹⁸⁹ Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors (in Finnish). Master's thesis, University of Helsinki, Finland, 1970

¹⁹⁰ Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, Jun 1976. ISSN 1572-9125. DOI: 10.1007/BF01931367

¹⁹¹ Paul John Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974. URL <https://www.researchgate.net/publication/35657389>

¹⁹² Paul John Werbos. Applications of advances in nonlinear sensitivity analysis. En R. F. Drenick y F. Kozin, editores, *System Modeling and Optimization: Proceedings of the 10th IFIP Conference New York City, USA, August 31 – September 4, 1981*, pages 762–770. Springer, 1982. ISBN 978-3-540-39459-4. DOI: 10.1007/BFb0006203

¹⁹³ David B. Parker. Learning-Logic: Casting the Cortex of the Human Brain in Silicon. Technical Report TR-47, Center for Computational Research in Economics and Management Science, Alfred P. Sloan School of Management, MIT, 1985

internas de una red neuronal multicapa.¹⁹⁴

Yann LeCun también derivó el algoritmo de *backpropagation* ese mismo año, de forma independiente.¹⁹⁵

El libro del grupo PDP [*Parallel Distributed Processing*], publicado también el mismo año, dio lugar al modelo conexionista como enfoque alternativo a la perspectiva simbólica de la I.A., entonces dominante. El libro establece los principios del conexionismo como forma de interpretar computacionalmente distintos aspectos de la cognición humana y del aprendizaje, basado en una representación distribuida que sitúa en las conexiones entre las neuronas el lugar de la memoria y del aprendizaje.¹⁹⁶

En 1993, Eric A. Wan fue el primero en conseguir ganar una competición internacional de reconocimiento de patrones utilizando *backpropagation*, entonces para predecir series temporales,¹⁹⁷ algo que se ha convertido en habitual hoy en día.

Durante los años 90, *backpropagation* gozó de cierta popularidad, aunque con el cambio de siglo las redes neuronales perdieron su popularidad en favor de otras técnicas de aprendizaje automático, como las máquinas de vectores de soporte SVM. En la segunda década del siglo XXI, las redes neuronales, ahora bajo la denominación de *deep learning* han recuperado la atención de los investigadores en I.A. gracias a la potencia computacional ofrecida por las GPGPU y la disponibilidad de grandes conjuntos de datos con los que entrenar redes neuronales mucho más grandes que las que se usaban en los años 90 del siglo pasado. Industrialmente, su uso se ha generalizado en dispositivos tan cotidianos como los *smartphones*, cuyos sistemas de reconocimiento de voz utilizan redes neuronales artificiales desde 2013. Además, también han conseguido sonados éxitos publicitarios en los medios de comunicación gracias a algunas hazañas que, hasta hace poco, estaban fuera del alcance de las técnicas de Inteligencia Artificial (p.ej. AlphaGo) y a su aplicación en aplicaciones de moda (p.ej. vehículos autónomos de Nvidia controlados por redes neuronales).

El entrenamiento de redes neuronales multicapa usando *backpropagation* y gradiente descendente, además de su peculiar historia y de sus múltiples redescubrimientos a lo largo de décadas, está relacionado con la investigación sobre *backpropagation* en redes neuronales biológicas, no artificiales.

El *backpropagation* neuronal es un fenómeno por el que el potencial de acción de una neurona crea un pulso eléctrico, un *spike*, que se propaga tanto a lo largo del axón (la propagación normal mediante la que se envían señales en el cerebro) como hacia atrás a través del árbol dendrítico (en dirección a las dendritas desde cuyas sinapsis llegó la señal de entrada que dio lugar al *spike*). Se ha observado experimentalmente que esta

¹⁹⁴ David E. Rumelhart, Geoffrey E. Hinton, y Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, October 1986a. DOI: 10.1038/323533a0

¹⁹⁵ Yann LeCun. Learning process in an asymmetric threshold network. En E. Bienenstock, F. Fogelman-Soulie, y G. Weisbuch, editores, *Disordered Systems and Biological Organization*, volume 20 de *NATO ASI Series*, pages 233–240, Les Houches, France, 1986. Springer. ISBN 978-3-642-82657-3. DOI: 10.1007/978-3-642-82657-3_24

¹⁹⁶ David E. Rumelhart, James L. McClelland, y the PDP Research Group, editores. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition - Volume 1: Foundations*. MIT Press, 1986b. ISBN 0262181207

¹⁹⁷ Eric Wan. Time Series Prediction by Using a Connectionist Network with Internal Delay Lines. En A. Weigend y N. Gershenfeld, editores, *Time Series Prediction: Forecasting the Future and Understanding the Past. SFI Studies in the Sciences of Complexity*, pages 195–217. Addison-Wesley, 1994

propagación hacia atrás existe en las neuronas biológicas, aunque se desconoce su función y hasta qué punto llega hasta las dendritas más distales (las que se encuentran más lejos del soma de la neurona), ya que la señal decae de forma significativa al propagarse por las dendritas. Las dendritas, a diferencia del axón, no están mielinadas, por lo que no propagan igual de bien las señales eléctricas y el *spike* hacia atrás pierde la mitad de su voltaje en apenas 500 micras.

Se ha especulado que este proceso podría aprovecharse en redes neuronales biológicas de la misma forma que el *backpropagation* de las redes neuronales artificiales multicapa. De hecho, el *backpropagation* neuronal aparece de forma más activa en regiones del cerebro de mayor plasticidad sináptica, como el neocórtex o el hipocampo, y de forma más pasiva en el cerebelo, que controla funciones subconscientes y vegetativas.

Algunas hipótesis sugieren que este fenómeno interviene en la plasticidad sináptica. Podría contribuir al aprendizaje hebbiano en el cerebro: un mecanismo básico de plasticidad sináptica en el que el valor de una conexión sináptica se incrementa si las neuronas de ambos lados de dicha sinapsis se activan repetidas veces de forma simultánea.¹⁹⁸

También podría estar involucrado en la potenciación a largo plazo [*LTP: Long-Term Potentiation*]: el fortalecimiento persistente de la fuerza de una sinapsis basado en patrones recientes de actividad, uno de los mecanismos clave asociados a la memoria y al aprendizaje, descubierto por el neurofisiólogo noruego Terje Lømo y el neurocientífico británico Timothy Bliss en 1966.¹⁹⁹ Su proceso opuesto es la depresión a largo plazo [*LTD: Long-Term Depression*], un tipo de plasticidad neuronal en el que se reduce la eficacia de una sinapsis. Por ejemplo, el aprendizaje hebbiano generalmente implica LTP en aquellas sinapsis con *spike* pre-sináptico ligeramente anterior al post-sináptico y LTD en el caso contrario.

Aunque el *spike* hacia atrás del *backpropagation* neuronal pueda, presumiblemente, contribuir a cambiar los pesos asociados a las conexiones presinápticas de una neurona, no está claro que exista un mecanismo biológico por el que la señal de error se propague a lo largo de múltiples capas de neuronas, como sucede con el *backpropagation* tradicional de las redes neuronales artificiales.

Intentando responder a las críticas sobre su escasa plausibilidad biológica, se han defendido interpretaciones locales de *backpropagation* con la intención de mostrar cómo las redes neuronales multicapa se entrena de forma que cada neurona sólo utiliza información “local” (que se propaga globalmente, habría que decir, desde la salida de la red hasta su entrada).²⁰⁰ Desde este punto de vista, una red multicapa se puede ver como si estuviese formada por tres tipos de neuronas: las unidades corticales, que se limitan a sumar sus entradas; las unidades sinápticas, que reciben una única entrada y aplican una función de activación; y, por último, las unidades talámicas, que comparan la salida de las unidades sinápticas de

El cerebelo se encarga principalmente del control de la precisión de los movimientos que realizamos. Resulta esencial para actividades motoras que requieren rapidez y precisión, como correr, escribir a máquina, tocar el piano o, incluso, hablar.

¹⁹⁸ Donald O. Hebb. *The Organization of Behavior*. John Wiley, New York, 1949

¹⁹⁹ Timothy Vivian Pelham Bliss y Terje Lomo. Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *The Journal of Physiology*, 232(2): 331–356, 1973. ISSN 1469-7793. DOI: 10.1113/jphysiol.1973.sp010273

²⁰⁰ Donald W. Fausett. Strictly local backpropagation. En *IJCNN'1990 International Joint Conference on Neural Networks*, pages III:125–130, June 1990. DOI: 10.1109/IJCNN.1990.137834

salida con el valor deseado, tras lo que generan una señal de error que desencadena el proceso de propagación hacia atrás. La topología resultante de una red así es equivalente a una red multicapa convencional en la que cada capa está compuesta de dos subcapas: una de unidades corticales (agregación) y otra de unidades sinápticas (función de activación). La salida de la red se conecta a la capa talámica, que genera una señal de error. Esta señal se propaga a través de la red, en la que alternan capas sinápticas y capas corticales, de la forma convencional. El resultado final es completamente equivalente a la implementación modular que hicimos del algoritmo de entrenamiento de redes neuronales multicapa usando *backpropagation* y gradiente descendente.

El gradiente descendente combinado con *backpropagation* no es la única forma posible de entrenar redes neuronales multicapa. De hecho, los algoritmos de aprendizaje del perceptrón se pueden extender a redes multicapa.²⁰¹ Utilizando una estrategia *greedy*, en la que se descompone el problema y se emplean algoritmos previos siempre que sea posible, se puede reducir el problema de entrenar una red multicapa al problema de entrenar una única neurona. De esta forma, se pueden lograr algoritmos más rápidos que el algoritmo de propagación de errores. Stephen Gallant, de la Northeastern University en Boston, propuso dos alternativas:

- Un método distribuido [*distributed method*] que usa una única capa con d neuronas ocultas cuyos pesos, fijos, se establecen aleatoriamente. De esta forma, las neuronas de salida tendrán $n + d$ entradas en lugar de las n entradas del problema. Las neuronas ocultas añadidas proporcionan una representación distribuida que hace más probable que los ejemplos sean linealmente separables, por lo que resultará más sencillo el proceso de aprendizaje del perceptrón. Este enfoque recuerda a las propuestas originales de Frank Rosenblatt para modelar la retina.
- Un algoritmo *greedy*, denominado algoritmo de la torre [*tower algorithm*], en el que se entrena una neurona a partir de las n entradas. A continuación, se entrena una segunda neurona usando $n + 1$ entradas, las n originales más la proporcionada por la neurona entrenada inicialmente, cuyos pesos se mantienen ahora fijos. Estas neuronas se pueden ir apilando formando una torre de neuronas, de ahí la denominación del algoritmo. Se puede demostrar que, para conjuntos de entrenamiento no contradictorios, una torre de $n + 1$ neuronas clasificará correctamente, al menos, un ejemplo de entrenamiento más que la torre de n neuronas que utiliza como base. El algoritmo propuesto por Stephen Gallant es, esencialmente, el mismo que dio lugar al nacimiento del *deep learning* en 2006, sólo que en 2006 se apilaron máquinas de Boltzmann o *autoencoders* en vez de neuronas individuales. De hecho, Gallant ya indicaba en su artículo de 1990 que en cada etapa

²⁰¹ Stephen I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, June 1990. ISSN 1045-9227. DOI: 10.1109/72.80230

se podrían utilizar máquinas lineales en vez de neuronas individuales, que es lo que se hizo 16 años después.

El propio Gallant reconoce, no obstante, que su algoritmo de la torre puede no ser particularmente eficiente ni garantizar una solución óptima, por lo que sugiere que su método distribuido puede resultar preferible. Hoy en día, salvo que tengamos motivos que lo justifiquen expresamente, no mantendríamos fijos los pesos de las neuronas ocultas, sino que los ajustaríamos con *backpropagation*.

En la misma época, dos investigadores franceses, Marc Mézard y Jean-Pierre Nadal, propusieron también otro algoritmo en el que el número de capas y neuronas de la red no se fijaba de antemano, sino que el algoritmo de aprendizaje era capaz de ir haciendo crecer la red hasta converger: el algoritmo del mosaico [*tiling algorithm*].²⁰² Algo similar a lo que hacía el método advenedizo [*upstart algorithm*] propuesto por Marcus Frean, de la Universidad de Edimburgo, que empleaba una versión modificada del algoritmo de aprendizaje del perceptrón para ir añadiendo unidades a la red de forma recursiva.²⁰³

Todas estas propuestas se pueden ver, en cierto modo, como intentos iniciales (rudimentarios, si se quiere) de encontrar técnicas automáticas que nos permitan establecer de forma adecuada los hiperparámetros asociados a la topología de una red neuronal artificial: sus dimensiones en cuanto a número de capas y número de neuronas por capa.

²⁰² Marc Mézard y Jean-Pierre Nadal. Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A: Mathematical and General*, 22(12):2191, 1989. URL <http://stacks.iop.org/0305-4470/22/i=12/a=019>

²⁰³ Marcus Frean. The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2(2):198–209, April 1990. ISSN 0899-7667. doi: 10.1162/neco.1990.2.2.198

Entrenamiento de una red neuronal

Desde el punto de vista formal, una red neuronal multicapa es un aproximador universal. En principio, una red de este tipo es capaz de aprender cualquier cosa, siempre que la red sea lo suficientemente grande como para representar las peculiaridades de la función que pretendemos aprender.

Si disponemos de un número suficiente de ejemplos de entrenamiento, una red multicapa de la capacidad suficiente será capaz de construir un modelo de los datos con los que la entrenamos. Por desgracia, no existe una definición formal de lo que resulta “suficiente”. El entrenamiento de redes neuronales para resolver problemas prácticos es, y tal vez lo sea siempre, más un arte que una ciencia. A lo máximo que podemos aspirar en la práctica es a descubrir una serie de criterios heurísticos que nos ayuden a tomar decisiones con respecto al diseño de una red neuronal y de su proceso de entrenamiento. Como todas las heurísticas, normalmente suelen ofrecer buenos resultados pero no garantizan nada desde un punto de vista formal.

A la hora de entrenar una red neuronal multicapa, hemos de tener en cuenta múltiples aspectos. Hemos de tomar decisiones con respecto tanto a los parámetros de diseño de la red como a los parámetros del algoritmo de entrenamiento que decidimos utilizar. Colectivamente, estos parámetros reciben el nombre de hiperparámetros para distinguirlos de los propios parámetros de la red, los pesos de sus sinapsis y los sesgos de sus neuronas que se ajustan durante el proceso de entrenamiento de la red.

¿Qué aspectos debemos considerar en el diseño y entrenamiento de una red neuronal artificial?

- *Topología de la red:* ¿Qué módulos utilizamos en la construcción de la red? ¿Cuántas capas ocultas? ¿Cuántas neuronas por capa? ¿Qué funciones de activación para las neuronas de las distintas capas?
- *Optimización:* ¿Qué modo de entrenamiento empleamos para ajustar los pesos de la red? ¿Cómo ajustamos los pesos de la red? ¿Con qué frecuencia? ¿Mantenemos fijas las tasas de aprendizaje? ¿Utilizamos la misma tasa de aprendizaje para todas las capas de la red? ¿Cómo inicializamos los pesos de la red?
- *Invarianza:* ¿Cómo conseguimos que la red sea robusta frente a transformaciones comunes en los datos?

- *Generalización:* ¿Cómo conseguimos que la red funcione bien con datos distintos a los del conjunto de entrenamiento?

En este capítulo, aprenderemos algunas heurísticas que suelen funcionar adecuadamente a la hora de ajustar los distintos hiperparámetros de una red neuronal artificial. Dichas heurísticas nos ayudarán a tomar decisiones relativas al diseño de una red y al diseño del proceso de entrenamiento necesario para que la red aprenda correctamente. Estas heurísticas deben interpretarse como consejos prácticos para la resolución de problemas mediante el uso de redes neuronales, recomendaciones que uno puede seguir (o no) cuando intenta afinar el rendimiento de un sistema de *deep learning*.

Existen algunas referencias de interés en las que se recoge lo que múltiples especialistas han aprendido sobre el entrenamiento de redes neuronales artificiales. Algo que sólo se suele aprender por medio de la experiencia y que suele ser un conocimiento tácito, no recogido de forma explícita en libros de texto y manuales al uso. Por ejemplo, existe una monografía exclusivamente dedicada a “trucos del oficio” [*tricks of the trade204 En dicha monografía se incluyen múltiples capítulos de interés. Entre ellos se encuentra uno de Yann LeCun sobre cómo entrenar de forma eficiente una red usando *backpropagation*.²⁰⁵ Yoshua Bengio también ofrece sus consejos con respecto al entrenamiento de redes utilizando algoritmos basados en el gradiente del error y cómo ajustar sus muchos hiperparámetros.²⁰⁶*

Más en general, Andrew Ng ofrece una serie de consejos prácticos sobre el uso de técnicas de aprendizaje automático en su curso de Stanford.²⁰⁷ Ng hace especial hincapié en la depuración y el diagnóstico de las implementaciones de técnicas de aprendizaje, para lo que resulta adecuado instrumentarlas para permitir su monitorización conforme vamos introduciendo cambios incrementales.

Como en cualquier proyecto de desarrollo de software, existen dos extremos a la hora de abordar un nuevo proyecto:

- En un extremo se encuentran las metodologías más formales, que promueven un diseño cuidadoso y muy detallado antes de comenzar la implementación. Estas metodologías dedican una gran cantidad de tiempo al diseño de la arquitectura del software y de los algoritmos que se emplearán. Se realiza un gran esfuerzo en la ingeniería de características, intentando determinar cuáles son exactamente las características más adecuadas para la resolución del problema que tengamos entre manos. Y, por supuesto, se intenta recopilar, de antemano, el conjunto de entrenamiento que nos permita resolver el problema. Si todo va bien (casi nunca es así), cuando implementemos los algoritmos necesarios, todo funcionará correctamente y habremos resuelto el problema.

Esta estrategia tiene la ventaja de permitir el diseño de nuevos al-

²⁰⁴ Grégoire Montavon, Genevieve B. Orr, y Klaus-Robert Müller, editores. *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700 of *Lecture Notes in Computer Science*. Springer, 2012. ISBN 364235288X. DOI: 10.1007/978-3-642-35289-8

²⁰⁵ Yann LeCun, Léon Bottou, Genevieve B. Orr, y Klaus-Robert Müller. Efficient backprop. En Grégoire Montavon, Genevieve B. Orr, y Klaus-Robert Müller, editores, *Neural Networks: Tricks of the Trade*, pages 9–48. Springer, 2nd edition, 2012. ISBN 364235288X. DOI: 10.1007/978-3-642-35289-8_3

²⁰⁶ Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. En Grégoire Montavon, Geneviève B. Orr, y Klaus-Robert Müller, editores, *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2nd edition, 2012a. ISBN 364235288X. DOI: 10.1007/978-3-642-35289-8_26

²⁰⁷ Andrew Yan-Tak Ng. Advice for Applying Machine Learning. *Stanford CS229 Machine Learning*, 2015. URL <http://cs229.stanford.edu/materials/ML-advice.pdf>

goritmos de aprendizaje, elegantes y escalables. Es la estrategia que debería seguir un investigador en aprendizaje automático.

- En la práctica, a menudo no están claras cuáles son las partes críticas del sistema que deseamos construir, dónde encontraremos mayores dificultades en el proyecto en el que nos embarcamos. Como en Ingeniería del Software, si tenemos un proyecto con un riesgo técnico potencialmente elevado, es normal que utilicemos una estrategia incremental de desarrollo, comenzando por los aspectos más arriesgados del sistema. De esta forma, obtenemos un prototipo funcional lo antes posible, con el que descubrir los factores clave que pueden determinar nuestro éxito o nuestro fracaso. Es, básicamente, lo que hacen las metodologías ágiles de desarrollo de software, promoviendo una realimentación que nos ayudará a tomar decisiones conforme avance nuestro proyecto.

Aunque tal vez no sea la estrategia más adecuada si deseamos contribuir al estado del arte diseñando nuevas técnicas de aprendizaje automático, es la mejor forma de resolver un problema práctico concreto lo más rápidamente posible.

Así pues, para el desarrollo de un proyecto de aprendizaje automático en general, o de *deep learning* en particular, la estrategia recomendada suele consistir en:

1. Establecer cuáles son las métricas mediante las que evaluaremos el rendimiento del sistema. Aunque la más habitual es la precisión (o alguna de sus variantes), puede que nos interese fijar un mínimo aceptable en cuanto a precisión (p.ej. la habitual de un operador humano) y evaluar nuestro progreso en función del grado de cobertura del sistema; esto es, el porcentaje de casos que se resuelven de forma automática, sin requerir la intervención manual de un operador humano. También puede de que estemos interesados en garantizar unos mínimos con respecto a la escalabilidad o el *throughput* del sistema, con el objetivo de dar soporte a una carga de trabajo determinada.
2. Poner en marcha un prototipo funcional completo del sistema lo antes posible. Este prototipo debe abarcar de extremo a extremo, desde la adquisición de datos de entrenamiento hasta el uso del sistema en su entorno de producción (o, al menos, en un entorno simulado que lo emule). Dicho prototipo, además, debe incluir los mecanismos de instrumentación adecuados que faciliten la monitorización del sistema y su diagnóstico para corregir errores e introducir mejoras.
3. Realizar cambios incrementales en el sistema, más o menos sobre la marcha, en función de los análisis que realicemos para determinar dónde se encuentran los principales problemas del sistema y cuáles son sus áreas potenciales de mejora.

El *throughput* de un sistema (que se suele traducir por ‘rendimiento’, pese a resultar ambiguo) mide la cantidad de trabajo que se realiza por unidad de tiempo.

Podemos analizar las métricas de rendimiento del sistema para intentar encontrarle una explicación a las diferencias que observemos entre el rendimiento actual del sistema y nuestros objetivos de rendimiento. Además de analizar resúmenes estadísticos de los experimentos que realicemos, podemos estudiar cómo funciona el sistema para casos particulares o la evolución del mismo durante su proceso de entrenamiento.

Además, también puede resultar de interés la realización de un análisis “ablativo”. El análisis ablativo consiste en ir eliminando componentes del sistema, uno a uno, para ver cuál es la contribución individual de cada uno de ellos. Este tipo de análisis nos permite estudiar las diferencias entre el rendimiento base de algún algoritmo simple [*baseline*] con el rendimiento real del sistema. Esto nos permite determinar las aportaciones de los diferentes componentes que vayamos añadiendo y de las decisiones individuales de diseño que vayamos tomando.

Una vez planteada nuestra estrategia general para resolver problemas de aprendizaje automático, pasemos a analizar con algo más de detalle los distintos aspectos del entrenamiento de una red neuronal sobre los que podemos tomar decisiones de diseño: su topología (capas y conexiones entre capas), sus elementos de procesamiento (p.ej. las funciones de activación de sus neuronas), el modo de entrenamiento de la red (por lotes, *online* o por minibatches), el preprocesamiento de los datos de entrenamiento, la inicialización de los parámetros de la red y el ajuste de las tasas de aprendizaje.

Topología de la red

Aunque la mayor parte de las redes neuronales que se utilizan en la práctica son simples redes multicapa, existe una gran variedad de bloques que se pueden utilizar en su construcción. Se pueden escoger capas de salida especializadas para determinados tipos de problemas, como las capas *softmax* utilizadas para problemas de clasificación. Se pueden seleccionar capas cuyos patrones de conectividad resulten adecuados para determinados tipos de aplicaciones, como las capas convolutivas utilizadas en procesamiento de imágenes. Se pueden utilizar módulos que doten de memoria a una red neuronal, construyendo redes recurrentes como las redes LSTM [*Long Short-Term Memory*].

Una vez establecida la arquitectura general de la red, el diseñador debe decidir el número de capas de la red, sus patrones de interconexión y el tamaño de cada una de las capas (esto es, el número de neuronas que forman parte de cada capa).

El abanico de posibilidades es extenso, lo que proporciona al diseñador

de redes neuronales múltiples grados de libertad a la hora de abordar un nuevo problema y una flexibilidad de la que carecen otras técnicas de Inteligencia Artificial. El diseño modular de los distintos bloques que pueden emplearse en la construcción de una red neuronal artificial nos ayudará a explorar múltiples opciones de diseño.

Aproximadores universales

Al estudiar el perceptrón, vimos cómo una red neuronal con una sola capa de parámetros ajustables es equivalente a un clasificador lineal. Es decir, sólo puede representar un pequeño subconjunto de funciones y es completamente incapaz de aprender a resolver problemas que no sean linealmente separables. Sin embargo, al añadir capas adicionales, una red neuronal multicapa es capaz de aproximar funciones continuas no lineales sobre subconjuntos compactos de \mathbb{R}^n , usando una sola capa oculta y un número finito de neuronas en dicha capa.

El teorema de aproximación universal establece que, dada una función de activación ϕ no constante, acotada y monótonamente creciente, podemos construir una función \hat{f} de la forma

$$\hat{f}(x) = \sum_{h=0}^k \alpha_h \phi \left(\sum_{i=0}^n w_{ij} x_i \right)$$

que sirva de aproximación a cualquier función continua f , con $\epsilon > 0$:

$$|f(x) - \hat{f}(x)| < \epsilon$$

Si nos fijamos en la expresión anterior, podemos observar que la función con la que pretendemos aproximar f tiene la misma forma que la función que describe la salida de una red neuronal multicapa con una capa oculta cuyas neuronas utilizan la función de activación ϕ y unidades lineales en la capa de salida.

En realidad, no se trata más que de una extensión del teorema de aproximación de Karl Weierstrass, de 1885, que luego generalizaría el norteamericano Marshall Harvey Stone en 1937, en lo que hoy se conoce como teorema de Stone-Weierstrass. Weierstrass aproximaba cualquier función continua en un intervalo cerrado con la ayuda de un polinomio, que se puede construir como una suma de polinomios de Bernstein.

Posteriormente, en 1957, el matemático ruso Andrey Kolmogorov demostró, para resolver el decimotercer problema de Hilbert, un teorema que establece que cualquier función continua definida sobre $[0, 1]^n$ puede aproximarse como una suma de la forma

$$\sum_{j=0}^{2n} h_j \left(\sum_{i=1}^n \psi_{ij}(x_i) \right)$$

Los polinomios de Bernstein sirven también de base a las curvas de Bézier, muy utilizadas en informática gráfica desde que el ingeniero francés Pierre Bézier las utilizó en 1962 para diseñar las carrocerías de los coches de Renault. En realidad, el matemático Paul de Casteljau ya las había desarrollado en 1959, trabajando para Citroën, otro fabricante francés de coches, pese a lo que siguen conociéndose con el nombre de curvas de Bézier.

En 1965, el matemático David Sprecher, de la Universidad de California, amplió el teorema de Kolmogorov y, ya en 1987, Robert Hecht-Nielsen usó el resultado de Sprecher para demostrar que cualquier función continua $f : [0, 1]^n \rightarrow R^m$ puede aproximarse con cualquier grado de precisión deseado utilizando una red con n neuronas de entrada, $2n + 1$ neuronas ocultas y m neuronas de salida.

Al año siguiente, en 1988, Ronald Gallant y Halbert White mostraron cómo una red con una sola capa oculta que utilizase una función coseno se podía ver como una red de Fourier, capaz de aproximar cualquier función continua como un caso particular de serie de Fourier, por lo que ya no se requerían entradas binarias, sino que se podían utilizar entradas reales.²⁰⁸ De hecho, a cualquier persona familiarizada con el análisis de Fourier, utilizado muy a menudo en procesamiento de señales, no le sorprenderá que una función como la definida arriba sirva de逼近ador universal. La transformada de Fourier representa una señal, en el tiempo como un sonido o en el espacio como una imagen, como la superposición de una serie de frecuencias constituyentes, lo que tiene múltiples aplicaciones prácticas. En el caso de una red neuronal, se representa como una superposición de los niveles de activación de las neuronas ocultas de la red, que permiten aproximar cualquier conjunto de entrenamiento que le proporcionemos a la red.

Ya en el contexto de redes neuronales multicapa, el teorema de aproximación universal fue demostrado inicialmente por George Cybenko, de la Universidad de Illinois en Urbana-Champaign, para funciones de activación sigmoidales.²⁰⁹ De forma independiente, el japonés Ken-ichi Funahashi publicó esencialmente el mismo resultado.²¹⁰ Ese mismo año, el austriaco Kurt Hornik, en colaboración con los norteamericanos Maxwell Stinchcombe y Halbert White, mostraron que también se podían emplear otras funciones de activación [*squashing functions*].²¹¹

Kurt Hornik, de la Universidad Técnica de Viena en Austria, también mostró que las derivadas de la red multicapa pueden aproximar las derivadas de la función aproximada por la red²¹² y que basta con que la función de activación sea continua, no constante y esté acotada para que una red neuronal multicapa sirva de un逼近ador universal.²¹³ Obviamente, la función de activación deberá incluir alguna no linealidad, si bien no es la elección de la función de activación sino la propia topología de la red la que le confiere sus propiedades de逼近ador universal a una red multicapa.

Formalmente, los resultados de Cybenko y Hornik establecen que una red multicapa con una única capa oculta puede aproximar cualquier función medible (también conocida como función de Borel o Borel-medible) para cualquier error $\epsilon > 0$ dado un número suficiente de neuronas ocultas. Las funciones de Borel son una clase más general que las funciones continuas: toda función continua es de Borel, pero no toda función de Borel es

²⁰⁸ A. Ronald Gallant y Halbert White. There exists a neural network that does not make avoidable mistakes. En *ICNN'1988 Proceedings of the IEEE 1988 International Conference on Neural Networks*, volume I, pages 657–664, July 1988. DOI: 10.1109/ICNN.1988.23903

²⁰⁹ George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989. ISSN 0932-4194. DOI: 10.1007/BF02551274

²¹⁰ Ken ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2(3):183 – 192, 1989. ISSN 0893-6080. DOI: 10.1016/0893-6080(89)90003-8

²¹¹ Kurt Hornik, Maxwell Stinchcombe, y Halbert White. Multilayer feed-forward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080. DOI: 10.1016/0893-6080(89)90020-8

²¹² Kurt Hornik, Maxwell Stinchcombe, y Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5): 551 – 560, 1990. ISSN 0893-6080. DOI: 10.1016/0893-6080(90)90005-6

²¹³ Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. ISSN 0893-6080. DOI: 10.1016/0893-6080(91)90009-T

continua. Así pues, las redes neuronales multicapa pueden interpretarse como aproximadores eficientes de funciones no lineales. Desde este punto de vista, se puede demostrar que, independientemente de la función que estemos intentando aprender, existe una red neuronal multicapa capaz de representarla. Nos lo dice el teorema de aproximación universal, demostrado por George Cybenko para funciones de activación sigmoidales en 1989, y sus posteriores extensiones.

Aunque los resultados originales se establecieron en términos de funciones de activación que se saturan para entradas netas muy positivas o muy negativas, como es el caso de las funciones sigmoidales, se han demostrado teoremas de aproximación universal para clases más amplias de funciones de activación. Por ejemplo, funciones de activación no polinómicas como la función de activación de las unidades lineales rectificadas [*ReLU: Rectified Linear Units*], muy común en *deep learning*.²¹⁴

A diferencia del teorema original de Weierstrass, no existe una demostración constructiva del teorema de aproximación universal. Su demostración nos indica que existe una red capaz de aproximar cualquier función medible, pero no cuál es esa red ni cómo obtenerla. Se trata de un teorema de existencia: sólo nos indica que existe una red neuronal multicapa capaz de aproximar cualquier función. Si utilizamos *backpropagation* y gradiente descendente, puede que el algoritmo de entrenamiento de la red no sea capaz de aprenderla. Incluso cuando estemos entrenando una red neuronal con la capacidad suficiente para aproximar una función dada (esto es, con el número de unidades ocultas necesario), el aprendizaje puede fallar por dos motivos:

- El algoritmo de optimización puede ser incapaz de encontrar los valores de los parámetros de la red que corresponden realmente a la función deseada: puede no converger si no establecemos adecuadamente sus hiperparámetros o, simplemente, quedarse atascado en un mínimo local de la función de error. El gradiente descendente, combinado con *backpropagation*, no garantiza encontrar el mínimo global de la función de error, sólo un mínimo local.
- El algoritmo de entrenamiento puede quedarse con la función equivocada si la red sobreaprende [*overfitting*] y se ajusta demasiado a su conjunto de entrenamiento, perdiendo entonces su capacidad de generalizar correctamente. Como nos recuerda el teorema de Wolpert, no existe un procedimiento universal que nos permita examinar un conjunto de entrenamiento específico y elegir una función que generalice correctamente para ejemplos que no estén en el conjunto de entrenamiento.

Un problema formal de todos los algoritmos de entrenamiento de redes neuronales basados en el gradiente es la posibilidad de que el algoritmo

²¹⁴ Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, y Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861 – 867, 1993. ISSN 0893-6080. DOI: 10.1016/S0893-6080(05)80131-5

de entrenamiento converja a un mínimo local de la función de error en el espacio de pesos. En la práctica, no obstante, no suele suponer un problema insalvable, ya que siempre podemos cambiar la topología de la red (u otros parámetros del algoritmo de entrenamiento) si el error observado con la configuración actual de la red es demasiado alto. Cuando la red obtiene una solución aceptable para nuestro problema, tampoco tenemos garantías de que sea la mejor solución posible: nunca sabremos si estamos en un mínimo local en vez de en el mínimo global de la función de error. Desde el punto de vista teórico, este hecho puede resultar poco satisfactorio (y es una fuente de críticas, no sólo a las redes neuronales, sino a muchas otras técnicas de Inteligencia Artificial). Desde el punto de vista práctico, no obstante, si conseguimos resolver de forma satisfactoria un problema, importa relativamente poco que se trate de un óptimo local o de que hayamos detenido el aprendizaje antes de alcanzar un mínimo real de la función de error.

Durante mucho tiempo, se pensó que el problema asociado a los mínimos locales en los que podría quedarse atascado el método del gradiente descendente con *backpropagation* era su principal limitación. Dado que no se garantiza encontrar el mínimo global de la función de error, sino sólo un mínimo local, y las funciones de error asociadas a las redes neuronales multicapa son funciones no convexas, el escepticismo se adueñó de muchos investigadores. Hoy en día, se argumenta que tal problema no existe en muchas aplicaciones prácticas, en las que la dimensionalidad de los datos es muy elevada y la presencia de mínimos locales es irrelevante en comparación con la presencia de otros puntos críticos de la función de error, los puntos de silla [*saddle points*].²¹⁵

Complejidad del aprendizaje

El teorema de aproximación universal nos dice que una red neuronal multicapa es capaz de representar un amplio conjunto de funciones, pero no nos dice nada acerca de cómo ajustar sus parámetros para aproximar la función deseada.

J. Stephen Judd, del Instituto Tecnológico de California (Caltech), analizó con detalle un problema directamente relacionado con el aprendizaje de redes neuronales, al que denominó problema de la carga [*loading problem*]. En forma de problema de decisión, el problema de la carga se puede enunciar de la siguiente forma: Dada una arquitectura de red multicapa y un conjunto de datos de entrenamiento, ¿existe una configuración de los parámetros de la red que permita que la red sea capaz de reconocer/representar todas las parejas (entrada, salida) del conjunto de datos de entrenamiento? Se puede demostrar que este problema es NP-completo.²¹⁶

Ronald Linn Rivest, famoso criptógrafo receptor del Premio Turing en

²¹⁵ Yann LeCun, Yoshua Bengio, y Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 5 2015. ISSN 0028-0836. DOI: 10.1038/nature14539

²¹⁶ J. Stephen Judd. *Neural Network Design and the Complexity of Learning*. MIT Press, 1990. ISBN 0262100452

2002 y la R en el sistema criptográfico de clave pública RSA, demostró, junto con Avrim Blum, entonces estudiante en el MIT, que el aprendizaje de los pesos de una red con n entradas y tres neuronas en dos capas es ya un problema NP-completo.^{217,218} Esto es, la complejidad del problema no es menor por muy simple que sea la red que utilicemos (en este caso, n neuronas de entrada, 2 neuronas ocultas y una única neurona de salida).

El checo Jirí Síma extendió el resultado de Blum y Rivest para el caso de neuronas sigmoidales: con dos neuronas sigmoidales ocultas y una neurona binaria con umbral de salida, el problema es NP-difícil.²¹⁹ De hecho, Sima también demostró que ajustar los pesos de una única neurona sigmoidal para minimizar el error cuadrático sobre el conjunto de entrenamiento es también un problema NP-difícil.²²⁰

Afortunadamente, se trata sólo de una limitación teórica que esquivaremos en la práctica diseñando algoritmos específicos para topologías concretas de red, si bien implica algunas consecuencias formales de interés acerca de lo que no debemos esperar de un algoritmo de entrenamiento de redes neuronales:

- Los resultados de complejidad son independientes de la funcionalidad proporcionada por las neuronas individuales de la red. Asumiendo que sólo nos interesan las redes cuyas neuronas incluyen algún tipo de no linealidad, para que no se puedan reducir a redes de neuronas lineales con una sola capa, ya sabemos que sólo con cambiar el tipo de los nodos de la red no seremos capaces de sortear la complejidad intrínseca del problema.
- La búsqueda de los parámetros de la red, realizada por cualquier algoritmo de entrenamiento que empleemos (por ideal que pueda ser), no puede escapar de la complejidad inherente al problema en sí. Aun para redes muy simples, el problema de decisión ya es NP-completo, por lo que el problema de determinar los valores necesarios para los distintos parámetros de la red será, al menos, tan difícil como el problema de decisión. Si $P \neq NP$, los algoritmos de aprendizaje de redes neuronales no pueden garantizar encontrar una solución óptima en tiempo polinómico y, como caso particular, el entrenamiento de una red basado en el gradiente descendente y *backpropagation* nunca será eficiente.
- El problema es intratable computacionalmente para redes multicapa de topología arbitraria, incluso aunque las limitemos a una única capa oculta con sólo dos neuronas. No obstante, se pueden establecer restricciones sobre la topología de la red que permitan que el problema sea resoluble en tiempo polinómico. Judd analiza el problema en términos de conos de soporte [*support cones*], que no son más que los conjuntos de nodos que pueden afectar al comportamiento de un

²¹⁷ Avrim Blum y Ronald L. Rivest. Training a 3-node neural network is NP-complete. *Neural Networks*, 5(1): 117–127, 1992. doi: 10.1016/S0893-6080(05)80010-3

²¹⁸ Avrim Blum y Ronald L. Rivest. Training a 3-Node Neural Network is NP-Complete. En D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 494–501. Morgan-Kaufmann, 1989. URL <https://goo.gl/J1dVHj>

²¹⁹ Jiri Sima. Back-propagation is not efficient. *Neural Networks*, 9(6):1017–1023, 1996. ISSN 0893-6080. doi: 10.1016/0893-6080(95)00135-2

²²⁰ Jiri Sima. Training a single sigmoidal neuron is hard. *Neural Computation*, 14(11):2709–2728, Nov 2002. ISSN 0899-7667. doi: 10.1162/089976602760408035

nodo de salida concreto (todos, habitualmente). Si representamos el solapamiento de los conos de soporte en forma de grafo, al que Judd denomina grafo SCI [*support cone interaction*], se puede establecer que la complejidad del problema está relacionada con la topología de dicho grafo, que sería un grafo completo en las redes utilizadas habitualmente. El problema seguiría siendo NP-completo aunque el grafo SCI fuese planar (cuando se pueda dibujar en un papel sin que se crucen sus aristas) o tuviese una estructura regular simple en forma de rejilla 2D. Sólo dejaría de ser NP-completo y pasaría a ser un problema resoluble en tiempo polinómico cuando su descomposición en árbol tuviese una anchura limitada [*armwidth*]. En cuanto dicha medida topológica aumentase con el tamaño de la red, el problema volvería a ser NP-completo. Curiosamente, determinar tal medida en un grafo es un problema NP-completo en sí mismo,²²¹ por lo que el resultado formal tampoco nos serviría de mucho en la práctica para determinar si una red con una topología particular puede entrenarse de forma eficiente. Se trata sólo de una construcción formal que nos indica cuál es la condición más débil que nos permitiría que el problema de aprender los parámetros de una red con una única capa oculta fuese tratable desde el punto de vista computacional.

Las redes neuronales tienen problemas para memorizar un conjunto de datos, algo que es trivial en un ordenador convencional (una máquina de Turing, si nos ponemos formales). De hecho, el problema conserva su complejidad computacional incluso aunque sólo queramos que la red sea capaz de memorizar el 67 % del conjunto de entrenamiento, ya que el problema sigue siendo NP-completo cuando el conjunto de entrenamiento contiene sólo 3 ejemplos y memorizar más de dos tercios de tres ejemplos es lo mismo que aprender los tres.

Dada una red neuronal multicapa con una topología determinada, el problema de memorizar un conjunto de entrenamiento en general es un problema demasiado difícil. Y eso sin entrar en si la red es capaz de generalizar correctamente más allá del conjunto de entrenamiento, que es lo que realmente nos interesa conseguir. Tener éxito a la hora de generalizar presupone, en cierta medida, la capacidad de memorizar de forma fiel y, por tanto, la capacidad de generalización es, al menos, tan difícil computacionalmente como la capacidad de memorización, que ya es intratable desde el punto de vista computacional.

Los resultados teóricos se refieren a redes con una sola capa oculta y, desafortunadamente, los investigadores que han trabajado en el tema han ignorado sistemáticamente el estudio de redes más profundas. ¿Por qué? Porque, para un número fijo de neuronas, el número de funciones diferentes que son computables por una red con una única capa oculta es mayor que el número de funciones computables por una red más profunda.

²²¹ Stefan Arnborg, Derek G. Corneil, y Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987. ISSN 0196-5212. DOI: 10.1137/0608024

Sin conocimiento previo alguno acerca de qué es lo que tiene que aprender la red, la estrategia óptima consiste en poner todas las neuronas posibles en una única capa oculta. Además, una implementación en hardware de la red sería, en principio, más rápida cuanto menor fuese su profundidad.

Imponer restricciones a lo que la red debe aprender y analizar formalmente la complejidad del problema de aprendizaje resultante puede parecer algo demasiado alejado de la realidad. Aunque se ha comprobado que incluso problemas de aprendizaje triviales (y, por ende, poco interesantes) siguen siendo intratables, esto no quiere decir que dichos resultados teóricos sean completamente inútiles. De entrada, nos indican que las restricciones impuestas, por muy artificiosas que parezcan, no son suficientes para evitar la complejidad intrínseca del problema que pretendemos resolver.

En cierto modo, tampoco debería sorprendernos demasiado la complejidad computacional del entrenamiento de redes neuronales. Todos los problemas de Inteligencia Artificial comparten esta propiedad. El objetivo del análisis de las capacidades y limitaciones de las redes neuronales artificiales es doble. Por un lado, puede ayudarnos a determinar en qué aplicaciones pueden ser más útiles las redes neuronales. Por otro, puede guiarnos en el diseño de la topología de la red más adecuada que nos permita resolver un problema práctico concreto.

La posibilidad de que existan algoritmos eficientes para buscar la configuración parcial correcta de una red se ha ignorado desde el punto de vista formal, si bien deja abierta la posibilidad de poder construir redes complejas a partir de módulos reutilizables, como si de piezas de Lego se tratase. Al fin y al cabo, no parece que nuestro cerebro necesite resolver problemas NP-difíciles para desenvolverse razonablemente bien en el mundo real.

Además, tampoco deberíamos olvidarnos de que nada es gratis, como nos recuerdan los teoremas “*no free lunch*” de David Wolpert, tanto en su versión original²²² como en su versión para problemas de optimización.²²³ No existe ningún algoritmo de aprendizaje que se pueda considerar, en general, superior a otro. Plantear un problema de aprendizaje como si se tratase de un problema de optimización, como se suele hacer en *deep learning*, no hace más sencillo el problema. Cualquier algoritmo de optimización A que se comporte en determinadas situaciones mejor que un algoritmo de optimización B, habrá otras ocasiones en las que se comporte peor que B.

Uno incluso se puede preguntar si, dentro de los problemas de optimización combinatoria, existen algunos que son intrínsecamente más difíciles que otros.²²⁴ Si consideramos la complejidad media de un problema de optimización particular para todos los posibles algoritmos, nos encontramos con un resultado similar al teorema de Wolpert: no hay problemas intrínsecamente más difíciles que otros (dentro de la clase de

²²² David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996. ISSN 0899-7667. DOI: 10.1162/neco.1996.8.7.1341

²²³ David H. Wolpert y William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, Apr 1997. ISSN 1089-778X. DOI: 10.1109/4235.585893

²²⁴ William G. Macready y David H. Wolpert. What makes an optimization problem hard? *Complexity*, 1(5):40–46, 1996. DOI: 10.1002/cplx.6130010511

problemas de optimización combinatoria que nos interesan). Ahora bien, si consideramos el nivel de dificultad de un problema para el algoritmo de optimización que resulte más adecuado para él, entonces sí hay problemas intrínsecamente más difíciles que otros. Se abre un pequeño rayo de esperanza. El secreto de nuestro éxito residirá, entonces, en ser capaces de encontrar la forma de formular un problema para que sea algo más tratable y descubrir un algoritmo que resulte óptimo para el problema de optimización, tal como se haya planteado.

Ya que carecemos de la posibilidad de garantizar matemáticamente que una técnica de aprendizaje sea la óptima, ¿cómo nos enfrentamos a problemas cuya evaluación ha de realizarse de forma empírica? Yann LeCun nos ofrece su respuesta al respecto:²²⁵ “*Hemos de darnos cuenta de que nuestras herramientas teóricas son muy débiles. En ocasiones, disponemos de buenas intuiciones matemáticas acerca de por qué una técnica particular debería funcionar. En ocasiones, nuestra intuición resulta incorrecta... Toda técnica razonable de aprendizaje automático ofrece algún tipo de garantía matemática... aunque no pueda utilizarse para ningún propósito práctico... Mientras tu método minimice algún tipo de función objetivo... se está sobre fundamentos teóricos sólidos...*”. Según LeCun, las preguntas que nos deberíamos hacer son “¿cómo de bien funciona un método concreto sobre un problema particular?” y “¿cuál es la clase de problemas para los que funciona bien?”.

Dado que, a menudo, resulta imposible llegar a conclusiones formales, no deberíamos rechazar explicaciones heurísticas por el simple hecho de ser poco rigurosas (o no estar avaladas empíricamente lo suficiente). Necesitamos utilizar heurísticas para inspirarnos y guiarnos a la hora de tomar decisiones. Tal como comenta Michael Nielsen, en la era de los descubrimientos, los exploradores hicieron grandes descubrimientos partiendo de creencias que eran erróneas en gran medida.²²⁶ Cristóbal Colón buscaba una ruta más corta hacia la India cuando descubrió accidentalmente América para la civilización occidental. Este tipo de errores es inevitable y necesario al adentrarnos en terreno desconocido. Con el tiempo, esos errores se van corrigiendo conforme se realizan nuevos descubrimientos y nuestro conocimiento sobre el tema se va ampliando. Como en otras épocas de descubrimientos, nuestro conocimiento actual sobre redes neuronales es aún muy deficiente, por lo que debemos ser audaces aun cuando no tengamos garantías rigurosas de que todos nuestros pasos sean siempre los correctos.

Tamaño de la red

El teorema de aproximación universal establece que una red neuronal multicapa con un número finito de neuronas es capaz de actuar como un aproximador universal, pero no nos dice cuántas neuronas serán

²²⁵ Yann LeCun. AMA - Ask Me Anything. reddit, 2014. URL <https://goo.gl/MK4H72>

²²⁶ Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL <http://neuralnetworksanddeeplearning.com/>

suficientes para que la red actúe como tal para un problema dado. Sólo nos indica que la red debe tener un número suficiente de neuronas ocultas. De hecho, en teoría, puede que nos haga falta un número exponencial de neuronas en la capa oculta para poder aproximar una función, una para cada posible configuración del vector de entrada.²²⁷ El resultado formal resulta fácil de interpretar en el caso binario: el número posible de funciones binarias definidas sobre vectores binarios $f : \{0, 1\}^n \rightarrow \{0, 1\}$ es 2^{2^n} ; por tanto, para ser capaces de seleccionar una de ellas, puede que necesitemos 2^n bits o, en general, $O(2^n)$ grados de libertad. Se trata de uno de esos casos que mencionaba LeCun, en los que la teoría nos proporciona garantías matemáticas sin utilidad práctica alguna, esta vez en forma de cota superior para el número necesario de neuronas ocultas.

En la práctica, por suerte para nosotros, muchas veces se consiguen resultados más que aceptables utilizando un número relativamente pequeño de neuronas. Al fin y al cabo, las redes neuronales reales no tienen que trabajar con datos arbitrarios elegidos por algún tipo de adversario que pretende obligarnos a necesitar un número exponencial de neuronas, sino que han de tratar con los datos que se presentan en la vida real. Sus regularidades muchas veces nos ayudarán a que una red relativamente pequeña sea capaz de resolver con éxito problemas de indudable interés práctico.

Dado un problema concreto, determinar el número más adecuado de neuronas ocultas no resulta sencillo, no obstante. Habitualmente, se procura minimizar el número de neuronas utilizado para reducir la carga computacional que supone el entrenamiento y uso de neuronas adicionales, ya que cada neurona de más añade una carga innecesaria a la CPU (o GPU) que estemos utilizando, además de incrementar el ancho de banda necesario para la transmisión de sus pesos y el espacio de almacenamiento necesario para almacenar la configuración de la red.

¿Cómo sabemos si nuestra red neuronal incluye un número suficiente de neuronas ocultas? Si el entrenamiento de la red no converge, posiblemente no consigamos el rendimiento deseado, por lo que es probable que tengamos que añadir más nodos ocultos a la red (o bien emplear un conjunto de entrenamiento mejor, que nos ayude a construir un modelo más preciso del problema que pretendemos abordar).

Si el entrenamiento de la red converge y, para nosotros, resulta fundamental reducir al máximo la carga computacional que requiere su uso, algo que sucede si pretendemos integrarla en alguna aplicación para dispositivos móviles alimentados por baterías, podemos intentar reducir el tamaño de la red. La idea es ir simplificando la red a la vez que monitorizamos su rendimiento, con el objetivo de que este último no se degrade demasiado. Por ejemplo, durante la fase de entrenamiento de la red, si detectamos que los pesos asociados a una neurona particular apenas cambian, puede que esa neurona no participe activamente en la

²²⁷ Andrew R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3): 930–945, May 1993. ISSN 0018-9448. DOI: 10.1109/18.256500

generación de las salidas de la red, por lo que tal vez resulte superflua y la podamos eliminar sin problemas. De hecho, se han propuesto innumerables técnicas de poda de redes neuronales basadas en heurísticas de este tipo.²²⁸

Profundidad de la red

Como acabamos de mencionar, una red neuronal con tres capas es suficiente para aproximar cualquier función, siempre que incluyamos un número suficiente de neuronas en la capa oculta y dispongamos de un conjunto de entrenamiento suficientemente completo.

No obstante, ya desde los años 80 y durante toda la década de los 90, muchos investigadores se dieron cuenta de que entrenar una red con una capa oculta no siempre resultaba idóneo, debido a las interacciones entre las neuronas de la única capa oculta.^{229,230} Con dos capas ocultas, una red se entrenaba mejor. Las neuronas de la primera capa oculta son capaces de extraer características locales. En cierto modo, particionan el espacio de entrada en regiones y extraen características de esas regiones. Las neuronas de la segunda capa oculta combinan esas características locales extraídas por las neuronas de la capa anterior y son capaces de identificar características globales, que suelen resultar más útiles para resolver el problema cuando se basan en las características ya extraídas por la primera capa oculta. Además, para determinados tipos de funciones, el número de neuronas necesario es menor cuando se utilizan dos capas ocultas en vez de una.

En definitiva, un problema resulta a menudo más fácil de resolver si se utiliza más de una capa oculta, donde más fácil significa que la red aprende más rápido. No obstante, el uso de múltiples capas ocultas, origen del término *deep learning*, presenta ciertos problemas prácticos que no se resolvieron de forma efectiva hasta el segundo renacimiento de las redes neuronales, allá por 2006. En los años 80 y 90, los investigadores intentaron emplear el gradiente descendente con backpropagation para entrenar redes profundas, pero sin demasiado éxito salvo en arquitecturas especializadas para determinadas aplicaciones. Las redes se podían entrenar, pero de forma muy lenta, demasiado como para resultar útiles en la práctica. Con las técnicas adecuadas de *deep learning*, que nos iremos encontrando más adelante, hoy en día se pueden entrenar redes neuronales con muchas capas ocultas para resolver problemas complejos.

En principio, para un mismo número de neuronas, una red neuronal con más de una capa oculta es capaz de representar menos funciones diferentes que una red que sólo tenga una capa oculta. Este hecho, junto con las dificultades asociadas al entrenamiento de redes profundas, hizo que, durante años, se optase por ampliar la red a lo ancho cuando se disponía de capacidad computacional adicional, añadiendo más neuronas

²²⁸ Jocelyn Sietsma y Robert J. F. Dow. Neural Net Pruning - Why and How. En *IEEE 1988 International Conference on Neural Networks*, volume 1, pages 325–333, July 1988. DOI: 10.1109/ICNN.1988.23864

En la página web del libro puede encontrar más referencias bibliográficas sobre el tema: [Entrenamiento > Hiperparámetros > Ajuste de la topología de la red](#).

²²⁹ Daniel L. Chester. Why two hidden layers are better than one. En *ICNN'1990 Proceedings of the 4th IEEE Annual International Conference on Neural Networks*, volume 1, pages 265–268. Lawrence Erlbaum, January 1990

²³⁰ Eduardo D. Sontag. Feedback stabilization using two-hidden-layer nets. *IEEE Transactions on Neural Networks*, 3(6):981–990, Nov 1992. ISSN 1045-9227. DOI: 10.1109/72.165599

Implementar una función booleana puede requerir $O(2^n)$ nodos en una capa oculta, con una cota inferior $\Omega(2^n/n^2)$, pero sólo $\Theta(\sqrt{2^n}/n)$ con múltiples capas ocultas.

Robert O. Winder. Threshold logic asymptotes. *IEEE Transactions on Computers*, C-19(4):349–353, April 1970. ISSN 0018-9340. DOI: 10.1109/TC.1970.222921

a la única capa oculta. Más recientemente, distintos investigadores han sido capaces de demostrar que existen familias de funciones que se pueden representar de forma compacta utilizando redes neuronales con varias capas ocultas pero que requerirían un número exponencial de neuronas en un red con una única capa oculta.²³¹ ²³² Esto nos sugiere la posibilidad de que las redes multicapa utilizadas en *deep learning* podrían ser capaces de realizar con un número polinómico de neuronas lo que requeriría un número exponencial de neuronas en una red con una sola capa oculta. No es algo que se haya conseguido demostrar formalmente pero los indicios apuntan a que, con la misma cantidad de recursos, una red con múltiples capas ocultas es capaz de implementar funciones de mayor complejidad y, por tanto, enfrentarse a problemas más difíciles.

¿Cuáles son las características de ese tipo de funciones, aquéllas para las que la profundidad adicional de una red resulta beneficiosa? Esencialmente, aquéllas que corresponden a problemas que se pueden descomponer de forma jerárquica. En el mundo real, resulta que las redes profundas [*deep*], con múltiples capas ocultas, suelen funcionar mejor que las redes poco profundas [*shallow*], con una única capa oculta. Este hecho parece venir avalado por la hipótesis de la variedad [*manifold hypothesis*]. Esta hipótesis sugiere que los datos multidimensionales con los que nos encontramos en el mundo real residen en variedades de dimensionalidad menor que el espacio multidimensional en el que nos los encontramos. Si recuerda el ruido blanco que se veía en una televisión analógica sin sintonizar, o cuando no recibía correctamente la señal de una emisión, sabrá que la nieve mostrada en pantalla no se parecía a una imagen real, de las que vemos habitualmente. Este ruido es el tipo de imagen que se obtiene si genera aleatoriamente un conjunto de píxeles. Difícilmente obtendrá nada parecido a la imagen de un objeto real. En otras palabras, las imágenes del mundo real residen en un subespacio d -dimensional del espacio n -dimensional correspondiente a los n píxeles que las forman, con d presumiblemente mucho menor que n . Este fenómeno explicaría por qué las redes con múltiples capas ocultas son capaces de aprender a resolver problemas complejos en el mundo real.

Así pues, la razón tras el éxito del *deep learning* es su capacidad para construir jerarquías de conceptos que se amoldan al tipo de fenómenos con los que nos encontramos en el mundo real. En cierto modo, las redes neuronales multicapa proporcionan mecanismos de abstracción similares a los que nos encontramos en los lenguajes de programación convencionales. Nos permiten definir módulos a partir de los que construir otros módulos más complejos. Las múltiples capas de la red nos permiten descomponer problemas complejos en sus partes constituyentes. Un problema complejo se descompone jerárquicamente en problemas más sencillos. Las primeras capas de la red resuelven cuestiones simples a partir de los datos de entrada. Las capas posteriores son capaces de construir conceptos más

²³¹ Olivier Delalleau y Yoshua Bengio. Shallow vs. Deep Sum-Product Networks. En J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *Advances in Neural Information Processing Systems 24*, pages 666–674. Curran Associates, Inc., 2011. URL <https://goo.gl/qE6Thi>

²³² Monica Bianchini y Franco Scarselli. On the complexity of neural network classifiers: A comparison between shallow and deep architectures. *IEEE Transactions on Neural Networks and Learning Systems*, 25(8):1553–1565, 2014. DOI: 10.1109/TNNLS.2013.2293637

Una variedad [*manifold*] es un objeto geométrico multidimensional que generaliza la noción intuitiva de curva en una dimensión y de superficie en dos dimensiones. Una variedad de dimensión d es un espacio que se parece localmente a \mathbb{R}^d .

Una red poco profunda, con una sola capa oculta, sería como un lenguaje de programación sin la capacidad de realizar llamadas a subrutinas, funciones o métodos.

complejos y abstractos (aprender representaciones, en la terminología habitual del área). Identificar una cara en una imagen es muy difícil si trabajamos directamente a nivel de píxeles. Sin embargo, es relativamente sencillo utilizar los píxeles para identificar fronteras (el término con el que se conocen los bordes en procesamiento de imágenes). A continuación, podemos combinar fronteras para localizar esquinas y segmentos de líneas continuas. Éstas nos pueden ayudar a delimitar formas geométricas simples, facetas de objetos en la imagen. Estas facetas conforman vistas bidimensionales de objetos tridimensionales. Una disposición particular de múltiples objetos en una escena nos sugiere su contenido. Su configuración particular, por ejemplo, nos permite diferenciar una cara de otra. Una red neuronal artificial, con sus múltiples capas, es capaz de emular el proceso completo que conduciría a la activación de la ‘neurona abuela’ en nuestro cerebro biológico, identificando una cara familiar en una imagen compleja formada por miles o incluso millones de píxeles.

Ante esto, nos surge la siguiente pregunta: ¿cuántas capas ocultas debería utilizar? Según Yoshua Bengio, la respuesta es sencilla. Simplemente, se van añadiendo capas ocultas hasta que el error sobre el conjunto de validación deje de disminuir (Bengio habla del error en el conjunto de prueba pero, si interpretamos la profundidad de la red como uno de sus hiperparámetros, la evaluación debe realizarse sobre el conjunto de validación para evitar que nuestra evaluación final sobre el conjunto de prueba esté sesgada).

Empíricamente, cuanto más profunda sea la red, mejor suele generalizar en múltiples aplicaciones prácticas como la identificación de objetos en imágenes o el reconocimiento de voz. Sin embargo, entrenar una red profunda no es algo que resulte exento de dificultades. Si entrenamos una red neuronal profunda con nuestro algoritmo de batalla, el gradiente descendente estocástico con *backpropagation*, posiblemente no obtengamos resultados mucho mejores que con una red neuronal poco profunda, de una sola capa oculta (si es que los obtenemos).

La causa que origina este problema ya lo mencionamos cuando analizamos el comportamiento del gradiente en las distintas capas de una red multicapa. Matemáticamente, los deltas de la capa c (gradientes del error con respecto a las entradas netas de las neuronas) los podríamos representar como:

$$\delta^c = \left(\prod_{k=c}^{C-1} D^k (W^{k+1})^\top \right) D^C \nabla_y E$$

donde D^k es una matriz diagonal con los valores de las derivadas de las funciones activación para las distintas neuronas de la capa k ; esto es, $f'(z^k)$. En la expresión anterior, W^k es la matriz de pesos de la capa k y $\nabla_y E$ es el gradiente del error con respecto a los niveles de activación de las neuronas de la capa de salida.

La expresión anterior nos indica que las diferentes capas de una red profunda aprenden a velocidades diferentes. Los factores de la forma $D^k(W^{k+1})^\top$, por lo general, harán que el gradiente sea menor para las capas de la red más lejanas a su salida. Aun cuando las últimas capas de la red estén aprendiendo adecuadamente, los gradientes que les llegan a las primeras capas puede ser tan pequeños que éstas se queden atascadas, sin aprender prácticamente nada, ya que los pesos asociados a sus conexiones apenas varían. El uso de neuronas sigmoidales, que se saturan, agrava el problema (las derivadas de su función de activación son prácticamente cero una vez saturadas las neuronas). Esto hace que las neuronas de las primeras capas de la red aprendan mucho más lentamente que las neuronas de las últimas capas. Es el problema de la desaparición del gradiente, también conocido como gradiente evanescente [*vanishing gradient*].

La desaparición del gradiente no es inevitable, pero su alternativa tampoco resulta nada halagüeña: la explosión del gradiente [*exploding gradient*]. Cuando el gradiente es mucho mayor en las primeras capas de la red y “explota”, el aprendizaje de la red tampoco resultará el adecuado.

En general, la aparición de múltiples factores $D^k(W^{k+1})^\top$ hace que el cálculo del gradiente resulte numéricamente inestable, por lo que tiende a desvanecerse (cuando esos factores son menores que 1) o explotar (cuando son mayores que 1). En otras palabras, la ralentización del aprendizaje en las capas que reciben un gradiente evanescente no es un mero accidente o una inconveniencia puntual, es un problema intrínseco del algoritmo con el que entrenamos la red, que se agrava cuantas más capas incluya ésta.²³³

El problema reside en que el gradiente de las capas anteriores de la red se obtiene como producto de los términos correspondientes a todas las capas posteriores. Si hay muchas capas, la situación es intrínsecamente inestable. La solución consistiría en conseguir que todas las capas aprendan más o menos a la misma velocidad, haciendo que todos esos términos se balanceen para que su producto sea, aproximadamente, de la misma magnitud. Es lo que hace, por ejemplo, la normalización por lotes [*batch normalization*]:²³⁴ normalizar los datos de cada minibatch antes de la entrada de cada capa, de forma que se reduzca el fenómeno que conduce a la inestabilidad del gradiente, conocido como “variación covariante interna” [*internal covariate shift*]. Sin mecanismos como éste, que estabilizan el aprendizaje, es muy improbable que un algoritmo simple basado en gradiente descendente y *backpropagation* logre entrenar con éxito una red neuronal con muchas capas ocultas.

¿Es la inestabilidad del cálculo del gradiente el único factor que dificulta el entrenamiento de redes neuronales artificiales con múltiples capas ocultas? Por desgracia, parece que no:

²³³ Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, y Jürgen Schmidhuber. Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies. En John F. Kolen y Stefan C. Kremer, editores, *A Field Guide to Dynamical Recurrent Neural Networks*, pages 237–244. IEEE Press, 2001. ISBN 0780353692

²³⁴ Sergey Ioffe y Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. En Francis Bach y David Blei, editores, *ICML'2015 Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR. URL <http://proceedings.mlr.press/v37/ioffe15.html>

- Xavier Glorot y Yoshua Bengio, de la Universidad de Montréal, encontraron pruebas de que el uso de funciones sigmoidales de activación dificulta el entrenamiento de redes profundas: las neuronas sigmoidales no simétricas con respecto al origen tienden a saturarse demasiado pronto durante el entrenamiento, lo que ralentiza sustancialmente el aprendizaje.²³⁵
- Ilya Sutskever, James Martens, George Dahl y Geoffrey Hinton, de la Universidad de Toronto, estudiaron el impacto que tiene sobre el aprendizaje la inicialización de los pesos y el uso de momentos en el gradiente descendente estocástico. Ambos aspectos influyen decisivamente en nuestra capacidad de entrenar correctamente redes neuronales profundas.²³⁶

Por tanto, además de introducir mecanismos que estabilicen el cálculo del gradiente, para entrenar una red neuronal con múltiples capas ocultas también deberemos prestar atención a la selección de la función de activación, a la estrategia de inicialización de los pesos de la red e, incluso, a detalles de cómo implementar el método de optimización basado en el gradiente descendente.

Antes de pasar a estudiar ese tipo de detalles, analicemos por qué, pese a las dificultades, puede interesarnos ser capaces de entrenar redes neuronales profundas. En 2014, el físico y matemático Guido Montúfar demostró que hay funciones representables con redes profundas que requieren un número exponencial de neuronas en una red con una única capa oculta.²³⁷ Más concretamente, mostró que redes con funciones de activación lineales a trozos (que incluyen a rectificadores ReLU y unidades *maxout*) pueden representar funciones con un número de regiones exponencial con respecto a la profundidad de la red.²³⁸ Esto es, aunque tanto las redes profundas como las redes poco profundas son aproximadores universales, hay funciones que una red de profundidad k puede representar eficientemente para las que una red de profundidad insuficiente necesitaría un número exponencial de neuronas ocultas (ya sea una red poco profunda con una única capa oculta o una red con profundidad $k - 1$).

Si conectamos este hecho con la hipótesis de la variedad [*manifold hypothesis*] y la maldición de la dimensionalidad [*curse of dimensionality*], podemos apreciar los beneficios formales que ofrecen las redes profundas. Las redes profundas son capaces de extraer características a un mayor nivel de abstracción que las redes poco profundas. Estas características se pueden interpretar como factores (o causas generadoras [*generative causes*]) que se obtienen de la composición de otros factores. La composición de múltiples no linealidades da lugar a un modelo que codifica la suposición de que las funciones que nos interesa aprender involucran la composición de otras funciones más simples.

²³⁵ Xavier Glorot y Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. En Yee Whye Teh y D. Mike Titterington, editores, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256, 2010. URL <http://jmlr.org/proceedings/papers/v9/glorot10a>

²³⁶ Ilya Sutskever, James Martens, George Dahl, y Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. En Sanjoy Dasgupta y David McAllester, editores, *ICML'13 Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/sutskever13.pdf>

²³⁷ Guido F. Montúfar. Universal approximation depth and errors of narrow belief networks with discrete units. *Neural Computation*, 26(7):1386–1407, July 2014. ISSN 0899-7667. DOI: 10.1162/NECO_a_00601

²³⁸ Guido F. Montúfar, Razvan Pascanu, Kyunghyun Cho, y Yoshua Bengio. On the number of linear regions of deep neural networks. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger, editores, *NIPS'2014 Advances in Neural Information Processing Systems 27*, pages 2924–2932. Curran Associates, Inc., 2014. URL <https://goo.gl/tGAZ46>

Desde el punto de vista del aprendizaje de representaciones, estamos descubriendo los factores de variación que explican el problema que pretendemos resolver, basándonos en otros factores de variación más simples. La topología de la red representa de forma explícita esa descomposición del problema, de la misma forma que un programa de ordenador se descompone en módulos más simples para abordar los subproblemas cuyas soluciones combinadas nos ayudan a resolver el problema original.

Desde el punto de vista de la representación distribuida de la información, proporcionada por los niveles de activación de las neuronas de una red neuronal según el modelo conexionista, estamos consiguiendo diferenciar un número exponencial de regiones en el espacio de los datos de entrada, utilizando para ello un número polinómico de parámetros.

Los clasificadores convencionales, como los basados en los vecinos más cercanos, árboles de decisión o máquinas de vectores de soporte SVM, utilizan representaciones no distribuidas. En una representación no distribuida, para un problema con entradas de d dimensiones y n características (prototipos en k-NN o k-means, hojas en un árbol de decisión, vectores de soporte en SVM), obtendremos un modelo con $O(nd)$ parámetros que nos ayudará a identificar $O(n)$ regiones diferentes, cada una con su conjunto de parámetros. Al utilizar la representación distribuida de una red profunda, los $O(nd)$ parámetros de nuestro modelo con n características (las neuronas de la red) nos permitirán diferenciar $O(n^d)$ regiones en el espacio de los datos de entrada.

Una representación no distribuida generaliza siempre localmente, asumiendo que entradas similares generan salidas similares [*smoothness assumption*]: si $u \approx v$, entonces $f(u) \approx f(v)$. Obviamente, es incapaz de aprender modelos complejos que correspondan a funciones en las que hay más picos y valles que ejemplos disponibles, por lo que sufre la maldición de la dimensionalidad.

En cambio, la representación distribuida aprendida por una red neuronal le permite generalizar a partir de atributos o características comunes, identificadas por las neuronas ocultas de la red. Esto ofrece una ventaja desde el punto de vista estadístico, ya que una función compleja puede representarse de forma compacta utilizando un número reducido de parámetros. Las regularidades de la función, más allá de la suposición de continuidad (si $u \approx v$, entonces $f(u) \approx f(v)$), permiten diferenciar tantas regiones como resulten de la intersección de n hiperplanos en \mathbb{R}^d :

$$\sum_{j=0}^d \binom{n}{j} \in O(n^d)$$

Así pues, si asumimos la hipótesis de la variedad, las redes neuronales profundas nos ofrecen ventajas a la hora de construir modelos. Con menos parámetros que ajustar, necesitaremos menos ejemplos para generalizar correctamente y esquivaremos la maldición de la dimensionalidad. Ade-

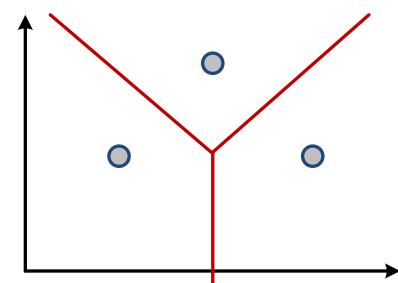


Figura 111: Regiones definidas por una representación no distribuida: Diagrama de Voronoi resultante de un clasificador k-NN o del clustering con k-means.

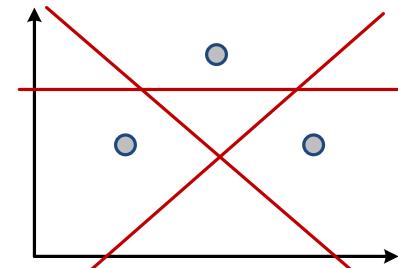


Figura 112: Regiones definidas por una representación distribuida: Los beneficios exponenciales de la profundidad en *deep learning*, donde n características definen n hiperplanos en \mathbb{R}^d y nos ayudan a diferenciar $O(n^d)$ regiones.

más, el proceso de entrenamiento de la red nos ayudará a determinar qué características son realmente relevantes, sin tener que establecerlas de antemano.

En resumen, el teorema de aproximación universal nos dice que, con redes neuronales multicapa, podemos aprender cualquier cosa. Aunque una única capa oculta sirva para conseguir un aproximador universal, esta capa podría necesitar un número enorme de neuronas (exponencial) y ser incapaz de generalizar correctamente. Usando capas adicionales, reducimos el número de neuronas necesario para representar la función deseada (que asumimos que se puede representar como composición de otras funciones más simples) y mejoramos la capacidad de generalización de nuestro modelo. En la práctica, esto funciona sorprendentemente bien (o, tal vez, de forma no tan sorprendente, si asumimos como cierta la hipótesis de la variedad).

Conejividad de la red

Antes de cerrar nuestras disquisiciones acerca de la topología de una red neuronal artificial, mencionaremos algunos apuntes acerca de cómo se pueden establecer conexiones entre las distintas capas de una red multicapa:

- *En cadena*: Lo más habitual en la práctica es conectar las capas formando una cadena, de forma que la salida de la capa k se utilice como entrada de la capa $k + 1$.
- *En cascada [cascade-forward networks]*: En vez de que la capa $k + 1$ reciba únicamente la salida de la capa k , se puede construir una red en la que cada capa reciba, como entrada, las salidas de todas las capas que la preceden en la red. Esto también sirve para acortar la longitud del camino desde la salida de una neurona hasta la salida de la red, lo que puede contribuir a mitigar los efectos derivados de la inestabilidad del cálculo del gradiente en redes multicapa.
- *Con saltos [skip connections]*: En muchas aplicaciones se parte de una arquitectura multicapa convencional, formada por una cadena de capas, a la que se añaden conjuntos de conexiones adicionales. Por ejemplo, se pueden añadir *skip connections* de la capa k a la capa $k + 2$. Estas conexiones adicionales, como en las redes en cascada, facilitan el flujo del gradiente desde las capas de salida hasta las capas más cercanas a la entrada, sin introducir tantos parámetros adicionales como los que requeriría una red en cascada.
- *Con múltiples cabezas [multi-headed]*: Otra posibilidad para facilitar el aprendizaje de una red multicapa con múltiples capas ocultas consiste en añadir nuevas capas de salida, que se conectan a capas intermedias

de la red. Estas capas de salida adicionales, denominadas cabezas, se utilizan sólo durante el entrenamiento de la red y, posteriormente, se descartan. De nuevo, la idea es mitigar los efectos derivados del gradiente evanescente, haciendo que todas las capas intermedias de la red reciban una señal de error que les permita aprender adecuadamente. Además, al introducir capas de salida intermedias, estamos obligando a que todas las capas de la red se ajusten de forma que contribuyan a resolver el problema para el que entrenamos la red.

- *Arquitecturas especializadas:* En algunas ocasiones, en vez de añadir nuevas conexiones, las eliminaremos. Por ejemplo, podemos hacer que cada unidad de una capa se conecte sólo a un subconjunto de unidades de la capa siguiente, como se hace en las redes convolutivas que se emplean para procesar imágenes. Reducir el número de conexiones reduce también el número de parámetros de la red y la capacidad de cálculo necesaria para su uso. La estrategia que se utilice para reducir el número de parámetros de la red depende del tipo de problema al que nos enfrentemos. A diferencia de las estrategias anteriores, que son genéricas, tendremos que diseñar una red específica para cada problema particular.

Funciones de activación

Dado que para entrenar una red neuronal utilizamos el gradiente del error y en el cálculo de dicho gradiente interviene la derivada de la función de activación, se suelen emplear funciones que sean derivables. Sin embargo, no es necesario que las funciones sean estrictamente derivables en todos los puntos. Suele bastar con que las funciones tengan definidas sus derivadas por la izquierda y por la derecha, aunque en algunos puntos no coincidan. Son habituales funciones en las que su derivada no está definida en algunos puntos, como la función rectificadora de las unidades lineales rectificadas, ReLU. Las implementaciones en software, simplemente, devuelven una de sus derivadas laterales, lo que no suele suponer problema alguno en la práctica. Al fin y al cabo, la optimización basada en el gradiente ya está sujeta a errores numéricos en su implementación digital, por lo que la no diferenciabilidad puntual de la función de activación no es algo que nos deba preocupar.

Esto, en principio, nos podría hacer pensar que se pueden utilizar funciones con alguna discontinuidad, como la función escalón, aunque no resultaría una buena idea, ya que su derivada (lateral en el punto del escalón) siempre es cero, lo que anularía el gradiente del error e imposibilitaría el entrenamiento de la red.

Funciones de activación sigmoidales

Xavier Glorot y Yoshua Bengio, de la Universidad de Montréal,²³⁹ encontraron pruebas de que el uso de funciones sigmoidales de activación dificulta el entrenamiento de redes profundas: las neuronas sigmoidales no simétricas con respecto al origen tienden a saturarse demasiado pronto durante el entrenamiento, lo que ralentiza sustancialmente el aprendizaje.

²³⁹ Xavier Glorot y Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. En Yee Whye Teh y D. Mike Titterington, editores, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256, 2010. URL <http://jmlr.org/proceedings/papers/v9/glorot10a>

De ahí que recomendasesen la tangente hiperbólica frente a la función logística y llegasen a proponer la función *softsign*, con un comportamiento asintótico más suave que la tangente hiperbólica.

La función *softsign* se define de la siguiente forma:

$$f_{\text{softsign}}(z) = \frac{z}{1 + |z|}$$

Su derivada se puede calcular de la siguiente forma:

$$\frac{df_{\text{softsign}}(z)}{dz} = \frac{d}{dz} \left(\frac{z}{1 + |z|} \right) = \frac{1}{(1 + |z|)^2}$$

Aunque la derivada de la función de activación decrece polinómicamente en lugar de exponencialmente, lo que la hace más interesante a la hora de evitar los problemas asociados a la saturación de la función de activación, es mucho más habitual el uso de la tangente hiperbólica como función de activación:

$$f_{\tanh}(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{1 - e^{-2z}}{1 + e^{-2z}}$$

En realidad, la tangente hiperbólica no es más que la versión bipolar de la función sigmoidal derivada de la función logística (con el parámetro de escala $s = 2$):

$$\tanh(z) = 2 \sigma(2z) - 1$$

Otra característica interesante de la tangente hiperbólica es que su derivada en un punto puede expresarse directamente en términos del valor de la función en ese punto, propiedad que puede aprovecharse para optimizar la implementación en software del cálculo del gradiente del error, reutilizando en la propagación del error hacia atrás el valor ya calculado para la función de activación de la neurona:

$$\frac{d \tanh(z)}{dz} = (1 + \tanh(z))(1 - \tanh(z)) = (1 - \tanh^2(z))$$

Las funciones sigmoidales simétricas con respecto al origen, como la tangente hiperbólica \tanh o la función *softsign*, permiten que el algoritmo de entrenamiento de la red converja más rápidamente, al evitar la saturación prematura de las neuronas sigmoidales de la red que se observa cuando se utiliza la función logística.

Si utilizamos la función logística en las capas ocultas de la red, todos los pesos de una neurona se actualizan en el mismo sentido. Si recordamos la función de actualización de los pesos del gradiente descendente estocástico, $\Delta w_{ij} = -\eta x_i \delta_j$, y observamos que la función logística de la que proviene la entrada x_i siempre toma valores positivos, podemos ver que, dependiendo del delta δ_j asociado a la neurona, o bien todos los pesos sinápticos de la neurona aumentan conjuntamente, o bien todos disminuyen. Esto

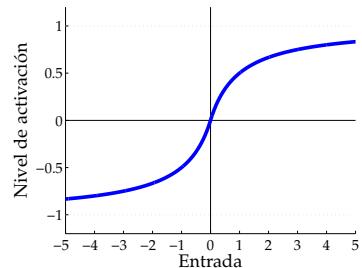


Figura 113: Función de activación *softsign*, más suave que la tangente hiperbólica.

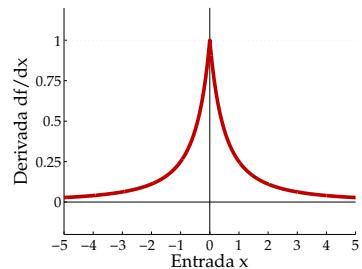


Figura 114: Derivada de la función *softsign*. Observe que la derivada decrece polinómicamente en lugar de exponencialmente, como sucede con la tangente hiperbólica.

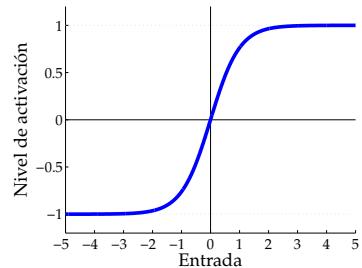


Figura 115: La tangente hiperbólica, simétrica con respecto al origen: $\tanh(-z) = -\tanh(z)$.

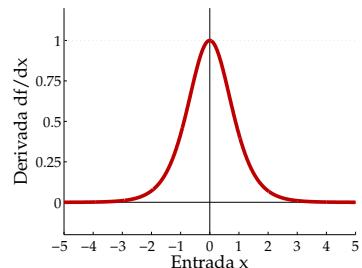


Figura 116: Derivada de la tangente hiperbólica. Obsérvese que su derivada es prácticamente cero antes que en el caso de la función *softsign*.

puede suponer un problema a la hora de entrenar la red, por lo que se recomienda el uso de funciones de activación bipolar, que toman valores tanto positivos como negativos.

Con respecto a la función logística, las funciones sigmoidales simétricas con respecto al origen tienen la ventaja adicional de producir niveles de activación con media cero. Estos niveles de activación, que se obtienen como salida de las neuronas ocultas y se utilizan como entradas en las capas siguientes de la red, les confiere a las neuronas sigmoidales simétricas con respecto al origen un efecto similar al causado por la normalización de los datos de entrenamiento en la capa de entrada de la red (usando *z-scores*). Los valores positivos de activación se equilibrarán con los valores negativos y no se introducirá ningún sesgo sistemático en la actualización de los pesos.

Así pues, tanto sus propiedades teóricas como las evidencias empíricas observadas sugieren el uso de funciones sigmoidales simétricas con respecto al origen en las capas ocultas de una red neuronal multicapa. La tangente hiperbólica suele funcionar mejor que la función logística.

Si se consigue evitar la saturación de las neuronas sigmoidales, que es lo que dificulta el entrenamiento basado en el gradiente del error, el entrenamiento de redes multicapa es similar al entrenamiento de un modelo lineal, mucho más eficiente. Para ello, hay que mantener pequeños los niveles de activación de las neuronas ocultas de la red, lo que resulta más fácil de conseguir si utilizamos la tangente hiperbólica.

En particular, Yann LeCun recomienda utilizar la siguiente función de activación:²⁴⁰

$$f(z) = 1.7159 \tanh\left(\frac{2z}{3}\right)$$

Se trata de una versión escalada de la tangente hiperbólica $a \tanh(bz)$, con $a = 1.7159$ y $b = 2/3$, seleccionada para que cumpla algunas propiedades deseables:

- La salida de la función coincide con su entrada para $z = \pm 1$: $f(\pm 1) = \pm 1$. Si utilizásemos la versión sin escalar de la tangente hiperbólica, la salida nunca llegaría a ± 1 , con lo que el error nunca se conseguiría eliminar (y, lo que es peor, siempre se observaría en el mismo sentido).
- En el origen, $z = 0$, la pendiente de la función de activación es también cercana a 1. En concreto: $f'(0) = 1.7159(2/3) = 1.1439$. Cuando se utiliza con entradas normalizadas usando *z-scores*, para que tengan media 0 y varianza 1, la ganancia efectiva de la neurona será cercana a 1 en todo su rango “útil”, en el intervalo $[-1, 1]$, en el que se comporta de forma prácticamente lineal.
- La segunda derivada de la función de activación alcanza su valor máximo en $z = -1$ y su mínimo en $z = +1$.

²⁴⁰ Yann LeCun, Leon Bottou, Genevieve B. Orr, y Klaus Robert Müller. Efficient BackProp. En Genevieve B. Orr y Klaus-Robert Müller, editores, *Neural Networks: Tricks of the Trade*, pages 9–50. Springer, 1998b. ISBN 3540653112. DOI: 10.1007/3-540-49430-8_2

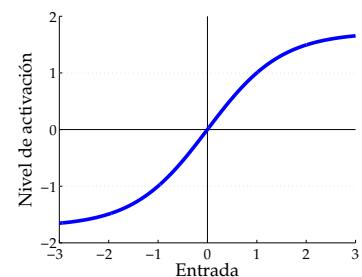


Figura 117: La tangente hiperbólica recomendada por Yann LeCun, con $a = 1.7159$ y $b = 2/3$. Observe cómo la función pasa por los puntos $(-1, -1)$, $(0, 0)$ y $(1, 1)$.

Los parámetros concretos de la tangente hiperbólica recomendada por LeCun se han seleccionado partiendo de la base de que los valores de salida deseados, con los que se entrena la red, son $y = \pm 1$. En problemas de clasificación, una neurona de salida se utilizaría para representar cada clase del problema. Para la neurona asociada a una clase particular, se esperaría obtener una salida $y = +1$ para los ejemplos pertenecientes a esa clase, $y = -1$ para los ejemplos que no pertenecen a ella.

Además, para evitar los problemas típicos asociados a la saturación de las neuronas sigmoidales, se amplía el rango de valores para los que las neuronas son sensibles a sus entradas, en torno a $z = 0$, de forma que la neurona nunca se sature en el intervalo $[-1, +1]$. Incluir una constante multiplicativa menor que 1, $b = 2/3$, nos permite jugar con la pendiente de la función de activación en la zona de interés y ampliar la zona en la que se comporta de forma lineal.

Para evitar problemas de saturación en las unidades de salida, se pueden elegir puntos en el rango de la sigmoide que maximicen su segunda derivada. De ahí surgió el parámetro $a = 1.7159$ de la recomendación de LeCun. No obstante, en la capa de salida de la red, el uso de la función logística es compatible con el aprendizaje basado en el gradiente descendente siempre que se utilice una función de coste apropiada, que deshaga la saturación de la función sigmoidal, como sucede en las redes *softmax* utilizadas para resolver problemas de clasificación.

En ocasiones, también resulta útil añadir un pequeño término lineal para evitar zonas planas en la función de activación (con gradiente nulo), p.ej.

$$f(z) = a \tanh(b z) + c x$$

Si, por algún motivo, decidimos ignorar las recomendaciones anteriores y utilizamos la función logística, cuya salida nunca puede llegar ni a 0 ni a 1, podemos utilizar valores como 0.1 o 0.9 para representar los valores mínimo y máximo de la salida deseada de la red. También podríamos, por ejemplo, desplazar verticalmente la función logística para que sus valores límite fuesen ± 0.4 , aunque eso no sería muy diferente de utilizar la tangente hiperbólica.

En cualquier caso, es importante que los valores objetivo (los que utilizamos para luego medir el error cometido por la red) estén siempre dentro del rango de valores que se obtienen como resultado de aplicar la función de activación sigmoidal. El valor asociado a la respuesta deseada de la red debería estar dentro de ese rango porque, si no es así, el error siempre tendría el mismo signo. Si el error tiene siempre el mismo signo, la corrección de los pesos siempre se hace en el mismo sentido y los parámetros de la red tenderán a infinito al entrenarla usando el gradiente descendente, lo que además fomentará la saturación de las neuronas y ralentizará el aprendizaje (defectuoso) de la red.

En realidad, para eliminar la saturación prematura de las unidades

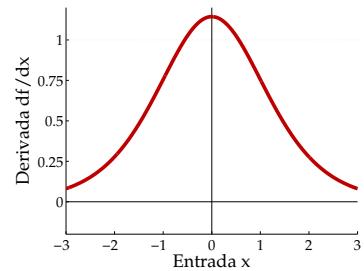


Figura 118: Derivada de la función de activación recomendada por Yann LeCun, cercana a 1 en el intervalo $[-1, 1]$.

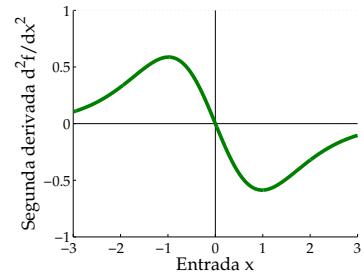


Figura 119: Segunda derivada de la función de activación recomendada por Yann LeCun, con óptimos en ± 1 .

sigmoidales se pueden utilizar múltiples estrategias, a menudo complementarias entre sí:

- *Inicialización de los pesos:*²⁴¹ Desde el punto de vista formal, la probabilidad de saturación de las neuronas aumenta conforme aumentan los valores máximos de los pesos. Estableciendo cuidadosamente sus valores iniciales podremos reducir el impacto que tienen sobre la saturación prematura de las neuronas sigmoidales.
- *Modificación de la fórmula de actualización de los pesos:* $\Delta w_{ij} = \eta x_i \delta_j$
 - *Término asociado al momento:*²⁴² El uso de momentos en la fórmula de actualización de los pesos contribuye a fomentar la saturación prematura de las neuronas sigmoidales. Cuando se detecta durante el aprendizaje, se puede reducir provisionalmente el factor α asociado al momento para eliminar la saturación prematura: $\Delta w_{ij}(t+1) = \eta x_i \delta_j + \alpha \Delta w_{ij}(t)$
 - *Término asociado a la derivada de la función de activación:*²⁴³ También se puede modificar el término que aparece en la fórmula de actualización de los pesos asociado a las derivadas parciales de la función de error, de forma que las señales de error se amplíen cuando las neuronas sigmoidales se aproximen a su región de saturación: $\Delta w_{ij} = \eta x_i \delta_j^{1/\rho}$, con $\rho \geq 1$.
 - *Término adicional relacionado con la saturación:*^{244,245} Igual que se añadía un momento para que las actualizaciones de los pesos conservasen cierta inercia en zonas planas de la superficie de error, se puede modificar la fórmula de actualización de los pesos utilizada en el gradiente descendente para que incluya un término adicional que tenga en cuenta el grado de saturación de las neuronas de la red y prevenga su saturación prematura.
- *Modificación de la función de activación:* Si utilizamos gradiente descendente con *backpropagation*, una red multicapa que utilice la función de activación logística σ , con pendiente o ganancia β , tasa de aprendizaje η , pesos W y sesgos b es matemáticamente equivalente a una red, de la misma topología, que utilice la función de activación σ con ganancia 1, tasa de aprendizaje $\beta^2 \eta$, pesos βW y sesgos βb . Este hecho nos permite jugar con los distintos parámetros de una función de activación sigmoidal:
 - *Ganancia adaptativa:* La ganancia de un nodo en una red es la pendiente de su función de activación. Este parámetro amplía o atenúa los cambios producidos en la entrada neta de la neurona.
 - Se puede hacer que la ganancia β de cada neurona sea un parámetro más de la red, ajustable con el gradiente descendente:^{246,247} $\Delta \beta_{ij} = -\eta_\beta \partial E / \partial \beta$. Este ajuste, en el fondo, equivale a utilizar tasas de aprendizaje adaptativas $\eta_{ij}(t)$.

²⁴¹ Y. Lee, S. H. Oh, y M. W. Kim. The effect of initial weights on premature saturation in back-propagation learning. En *IJCNN'91 Proceedings of the IEEE International Joint Conference on Neural Networks, Seattle, WA.*, volume 1, pages 765–770, Jul 1991. DOI: 10.1109/IJCNN.1991.155275

²⁴² Javier E. Vitela y Jaques Reifman. Premature saturation in backpropagation networks: Mechanism and necessary conditions. *Neural Networks*, 10(4):721–735, June 1997. ISSN 0893-6080. DOI: 10.1016/S0893-6080(96)00117-7

²⁴³ S.C. Ng, S.H. Leung, y A. Luk. Fast convergent generalized backpropagation algorithm with constant learning rate. *Neural Processing Letters*, 9(1):13–23, Feb 1999. ISSN 1573-773X. DOI: 10.1023/A:1018611626332

²⁴⁴ Hahn-Ming Lee, Tzong-Ching Huang, y Chih-Ming Chen. Learning efficiency improvement of back propagation algorithm by error saturation prevention method. En *IJCNN'99 Proceedings of the IEEE International Joint Conference on Neural Networks*, volume 3, pages 1737–1742 vol.3, 1999. DOI: 10.1109/IJCNN.1999.832639

²⁴⁵ Hahn-Ming Lee, Chih-Ming Chen, y Tzong-Ching Huang. Learning efficiency improvement of back-propagation algorithm by error saturation prevention method. *Neurocomputing*, 41(1):125–143, 2001. ISSN 0925-2312. DOI: 10.1016/S0925-2312(00)00352-0

²⁴⁶ J. K. Kruschke y J. R. Movellan. Benefits of gain: Speeded learning and minimal hidden layers in back-propagation networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(1):273–280, Jan 1991. ISSN 0018-9472. DOI: 10.1109/21.101159

²⁴⁷ Alessandro Sperduti y Antonina Stafita. Speed up learning and network optimization with extended back propagation. *Neural Networks*, 6(3):365–383, 1993. ISSN 0893-6080. DOI: 10.1016/0893-6080(93)90004-G

- El ajuste de la ganancia también puede realizarse utilizando una función de activación sigmoidal de la forma $f(z) = \sigma^\beta(z)$, con $\beta > 0$. Esta función de activación permite que el algoritmo de entrenamiento resultante sea más rápido que el estándar.²⁴⁸
- Por poder, se puede recurrir hasta a la lógica difusa para diseñar un mecanismo de ajuste automático de la ganancia de la función de activación.²⁴⁹ Este mecanismo, como los anteriores, se basa en criterios meramente heurísticos, sin justificación formal sólida.
- *Ampliación de la zona lineal de la neurona sigmoidal:* Se pueden diseñar funciones de activación que se comporten de forma similar a una función logística pero dispongan de una zona lineal más amplia. Aunque esto es algo que ya hemos visto que se puede conseguir fácilmente, introduciendo un factor que cambie la pendiente de la sigmoide en su zona lineal, las funciones que se han llegado a proponer pueden llegar a extremos casi barrocos, como en la siguiente función *log-exp*:²⁵⁰

$$f_{\text{logexp}}(z) = \frac{1}{2\beta b} \ln \left(\frac{1 + e^{\beta(z+b)}}{1 + e^{\beta(z-b)}} \right)$$

que se aproxima a la función logística $1/(1 + e^{-\beta z})$ cuando el parámetro $b \rightarrow 0$. En este caso, β determina la anchura de la zona lineal y b la pendiente en $z = 0$.

- *Función de activación con homotopía:*²⁵¹ Se define una función de activación de la forma $f(z) = \mu z + (1 - \mu)\sigma(z)$, con un componente lineal y otro sigmoidal, que iremos variando durante el entrenamiento de la red. Al principio, $\mu = 1$ y las neuronas se comportan de forma lineal, lo que nos permite encontrar rápidamente un mínimo local de la función de error. A continuación, se va disminuyendo el valor de μ hasta que $\mu = 0$, consiguiendo un proceso de entrenamiento similar al enfriamiento simulado, una de las metaheurísticas más conocidas para resolver problemas complejos de optimización
- *Modificación de la función de error:* Una cuarta estrategia consiste en diseñar una función de error que se dispare en situaciones en las que las neuronas se saturan (cuando las derivadas de la función de activación tienden a cero). De esta forma, se consiguen que las actualizaciones de los pesos siempre sean finitas pero no nulas, con lo que no se atasca el proceso de aprendizaje de la red.

- La modificación de la función de error puede consistir en añadir un simple término cuadrático adicional:²⁵²

$$E = \frac{1}{2}(t - y)^2 + \frac{\lambda_c}{2}(z - f^{-1}(y))^2$$

donde λ_c es un número positivo pequeño ($0 \leq \lambda_c \leq 1$) y $f^{-1}(y)$ es la inversa de la función de activación $f(z)$. En la práctica, esta

²⁴⁸ Pravin Chandra y Yogesh Singh. An activation function adapting training algorithm for sigmoidal feedforward networks. *Neurocomputing*, 61: 429 – 437, 2004. ISSN 0925-2312. DOI: 10.1016/j.neucom.2004.04.001. Hybrid Neurocomputing: Selected Papers from the 2nd International Conference on Hybrid Intelligent Systems

²⁴⁹ Kihwan Eom, Kyungkwon Jung, y Harsha Sirisena. Performance improvement of backpropagation algorithm by automatic activation function gain tuning using fuzzy logic. *Neurocomputing*, 50:439–460, 2003. ISSN 0925-2312. DOI: 10.1016/S0925-2312(02)00576-3. URL <http://www.sciencedirect.com/science/article/pii/S0925231202005763>

²⁵⁰ Włodzisław Duch. Uncertainty of data, fuzzy membership functions, and multilayer perceptrons. *IEEE Transactions on Neural Networks*, 16(1):10–23, Jan 2005. ISSN 1045-9227. DOI: 10.1109/TNN.2004.836200

²⁵¹ Liping Yang y Wanzen Yu. Backpropagation with homotopy. *Neural Computation*, 5(3):363–366, May 1993. ISSN 0899-7667. DOI: 10.1162/neco.1993.5.3.363

²⁵² S. Abid, F. Fnaiech, y M. Najim. A fast feedforward training algorithm using a modified form of the standard backpropagation algorithm. *IEEE Transactions on Neural Networks*, 12(2):424–430, Mar 2001. ISSN 1045-9227. DOI: 10.1109/72.914537

modificación consigue que el algoritmo converja más rápidamente. Además, podemos ir variando el valor de λ_c durante el entrenamiento de la red, haciendo que disminuya desde 1 hasta 0.

- En la función de error se puede incluir también un término que represente el grado de saturación de todas las neuronas de la red:²⁵³

$$E = \frac{1}{2}(t - y)^2 \left(1 + \sum_h (y_h - 0.5)^2 \right)$$

donde los y_h representan los niveles de activación de las neuronas ocultas y 0.5 el valor medio de la función de activación sigmoidal (asumiendo que estamos utilizando la función logística).

- También podemos definir funciones de error a medida para otras funciones de activación, como la tangente hiperbólica.²⁵⁴ Cuando se saturan las neuronas de salida de la red, proporcionando una salida incorrecta, una función de error que deshaga la saturación, como la entropía cruzada utilizada en las redes *softmax* que se emplean en problemas de clasificación puede solucionar el problema. Aun cuando la salida sea correcta, se puede generar una señal “débil” de error que facilite la perturbación de los pesos de la red y mejore su entrenamiento.

Como hemos visto, disponemos de múltiples grados de libertad que nos permiten ajustar el proceso de entrenamiento de una red con el objetivo común de tratar de evitar la saturación prematura de las neuronas sigmoidales, causa última tras la que se encuentran muchos de los problemas de aprendizaje asociados al entrenamiento de redes multicapa con gradiente descendente y *backpropagation*. Un ajuste adecuado de todos esos grados de libertad depende, en gran medida, de la experiencia previa que uno tenga. Por desgracia, la estrategia más adecuada para establecer los múltiples parámetros que nos ofrece el entrenamiento de una red con neuronas sigmoidales dependerá del problema concreto que deseemos resolver. Poco más se puede decir desde un punto de vista objetivo, más allá de un “hágalo a ojo de buen cubero”.

Unidades lineales rectificadas (ReLU)

En *deep learning*, es frecuente el uso de unidades lineales rectificadas [*ReLU: Rectified Linear Units*].

$$f_{relu}(z) = z^+ = \max(0, z)$$

Las neuronas lineales rectificadas se limitan a realizar una combinación lineal de sus pesos y entradas, tras lo que generan una salida no lineal utilizando un umbral, de ahí que también reciban el nombre de neuronas lineales con umbral [*linear threshold neurons*].

²⁵³ X.G. Wang, Z. Tang, H. Tamura, y M. Ishii. A modified error function for the backpropagation algorithm. *Neurocomputing*, 57:477 – 484, 2004. ISSN 0925-2312. DOI: 10.1016/j.neucom.2003.12.006. New Aspects in Neurocomputing: 10th European Symposium on Artificial Neural Networks 2002

²⁵⁴ Sang-Hoon Oh. Improving the error backpropagation algorithm with a modified error function. *IEEE Transactions on Neural Networks*, 8(3):799–803, May 1997. ISSN 1045-9227. DOI: 10.1109/72.572117

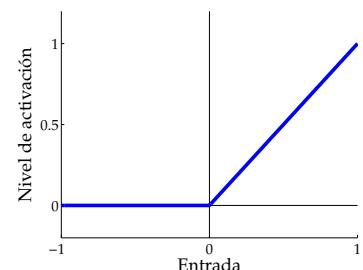


Figura 120: Función de activación lineal rectificada, utilizada por las unidades ReLU habituales en *deep learning*.

Al no requerir el uso de funciones trascendentales, como la exponenciación de la función logística o de la tangente hiperbólica, su evaluación es mucho más eficiente y el entrenamiento de redes neuronales con unidades ReLU suele ser mucho más rápido que con neuronas sigmoidales.

Además, un modelo numérico, como en el fondo es una red neuronal artificial, generalmente es más sencillo de optimizar cuando su comportamiento es lineal, lo que sucede siempre que se utilizan unidades ReLU. Cuando se activa, una neurona ReLU es equivalente a una simple neurona lineal. Cuando no se activa, permanece completamente silente y no interfiere en el comportamiento del resto de la red.

Aunque las neuronas sigmoidales de tipo logístico sean algo más biológicamente plausibles que las neuronas que utilizan la tangente hiperbólica, éstas últimas funcionan mejor que las logísticas en la práctica. Las neuronas lineales rectificadas, ReLU, pueden funcionar mejor aún, pese a su no linealidad y no diferenciabilidad en $z = 0$. Desde el punto de vista biológico, tal vez sean incluso más plausibles que las neuronas sigmoidales logísticas, al fomentar una representación dispersa de la información.²⁵⁵ La actividad realmente nula de las neuronas ReLU es algo que también se observa en las redes neuronales biológicas, una característica esencial de un cerebro biológico que le permite ahorrar energía y funcionar de forma más eficiente. El diseño de una unidad ReLU pretende capturar tres propiedades observadas en las neuronas biológicas: para algunas entradas, se mantienen completamente inactivas (cuando $z \leq 0$); para otras entradas, su salida es proporcional a su entrada (comportamiento lineal); y, por último, la mayor parte del tiempo, se mantienen inactivas (activaciones dispersas).

Sin embargo, la principal limitación de las unidades ReLU es que son incapaces de aprender cuando su nivel de activación es nulo, al ser su función de activación completamente plana en esta zona. Por este motivo, conviene inicializar los sesgos de las neuronas ReLU con un valor positivo pequeño (p.ej. $b = 0.1$), para hacer probable que las neuronas se activen durante las etapas iniciales del entrenamiento de la red.

Además, se han propuesto algunas generalizaciones de las neuronas ReLU con el objetivo de facilitar su entrenamiento, aunque sea a costa de prescindir de su plausibilidad biológica. Usualmente, la función de activación se modifica para que tenga una pendiente no nula (α) para valores negativos de la entrada neta a la neurona, z :

$$y = f_{rectification}(z) = \max(0, z) + \alpha \min(0, z)$$

- La rectificación de valor absoluto utiliza $\alpha = -1$ y se emplea en situaciones en las que no importa la polaridad de la señal de entrada, como en el reconocimiento de objetos en imágenes:²⁵⁶

$$y = f_{absolute\ rectification}(z) = \max(0, z) - \min(0, z) = |z|$$

²⁵⁵ Xavier Glorot, Antoine Bordes, y Yoshua Bengio. Deep sparse rectifier neural networks. En Geoffrey Gordon, David Dunson, y Miroslav Dudík, editores, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *JMLR W&CP, Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL <http://proceedings.mlr.press/v15/glorot11a>

²⁵⁶ Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, y Yann LeCun. What is the best multi-stage architecture for object recognition? En *IEEE 12th International Conference on Computer Vision, ICCV 2009, Kyoto, Japan, September 27 - October 4, 2009*, pages 2146–2153, 2009. ISBN 978-1-4244-4420-5. DOI: 10.1109/ICCV.2009.5459469

- Las unidades ReLU con pérdidas [*leaky ReLU*] utilizan un pequeño valor de α , p.ej. 0.01, sólo para evitar una derivada nula y permitir el ajuste de los pesos durante el entrenamiento de la red.²⁵⁷

$$y = f_{\text{leaky relu}}(z) = \max(0, z) + 0.01 \min(0, z)$$

- Las unidades ReLU paramétricas, o PReLU [*Parametric ReLU*], interpretan la pendiente α como un parámetro más de la neurona, que hay que entrenar como los pesos y puede ser diferente para cada neurona de la red:²⁵⁸

$$y_j = f_{\text{prelu}}(z, \alpha_j) = \max(0, z) + \alpha_j \min(0, z)$$

Como sucedía con las funciones de activación logísticas, pueden diseñar generalizaciones de las unidades lineales rectificadas, como es el caso de las unidades *maxout*. Una unidad de este tipo agrupa sus entradas en subgrupos y combina las salidas para cada subgrupo, lo que equivale a utilizar múltiples conjuntos de pesos en cada neurona:²⁵⁹

$$y_{\text{maxout}} = \max_k \{z_k\} = \max_k \{w_k \cdot x\}$$

Tanto las unidades ReLU convencionales como las *leaky ReLU* se pueden considerar casos particulares de este tipo de unidades, usando dos conjuntos de pesos: $(w_1, w_2) = (0, w)$ para las unidades ReLU y $(w_1, w_2) = (w, -\alpha w)$ para las *leaky ReLU*. Las neuronas *maxout*, por tanto, gozan de las ventajas de las unidades ReLU (funcionamiento lineal, sin saturación), sin tener sus inconvenientes (activación nula que dificulta su entrenamiento), a costa de duplicar su número de parámetros, claro.

En *deep learning*, ¿qué tipo de neurona se emplea realmente? Depende mucho de las tradiciones de cada equipo de trabajo, aunque a menudo se usan unidades de tipo ReLU. Si, al monitorizar el funcionamiento de la red, observamos que demasiadas neuronas se mantienen completamente inactivas la mayor parte del tiempo (las denominadas unidades muertas [*dead units*]), entonces podemos recurrir a unidades *leaky ReLU* o *maxout*.

Muchos evitan las unidades sigmoidales en la práctica y, cuando las usan, emplean la tangente hiperbólica en vez de la función sigmoidal, aunque suelen sugerir que su rendimiento esperado es peor que el de las unidades ReLU o sus generalizaciones más recientes, como las unidades *maxout*. Xavier Glorot, Antoine Bordes y Yoshua Bengio defienden que el entrenamiento de una red neuronal profunda con unidades lineales rectificadas resulta más sencillo que el de redes neuronales cuyas funciones de activación se saturan. Kevin Jarrett y sus colaboradores del grupo de Yann LeCun llegan a afirmar que el uso de unidades rectificadas es el factor más importante a la hora de mejorar el rendimiento de una red neuronal, de las distintas decisiones que uno puede tomar en el diseño de la arquitectura de una red neuronal artificial.

²⁵⁷ Andrew L. Maas, Awni Y. Hannun, y Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. En *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013. URL <https://goo.gl/5x3Cj6>

²⁵⁸ Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv e-prints*, 2015a. URL <http://arxiv.org/abs/1502.01852>

²⁵⁹ Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, y Yoshua Bengio. Maxout networks. En *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1319–1327, 2013b. URL <http://jmlr.org/proceedings/papers/v28/goodfellow13.html>

Desde el punto de vista algorítmico, uno de los principales cambios que se han producido en las redes neuronales artificiales, desde los años 80 hasta la moda actual del *deep learning*, es la sustitución de las funciones de activación sigmoidal por funciones de activación lineales a trozos, como las unidades ReLU. La idea de rectificar la salida de una neurona no es realmente novedosa, sino que se retrotrae hasta los años 70, con el Cognitrón y el Neocognitrón de Fukushima, si bien entonces no se utilizaban unidades lineales rectificadas, sino que se rectificaba la salida de unidades no lineales en modelos cuyo funcionamiento se inspiraba en las neuronas bipolares de la retina, las cuales generan dos señales rectificadas a partir de una imagen proyectada sobre la retina (una para las zonas claras y otra para las zonas oscuras). En los años 80, la rectificación desapareció prácticamente de las redes neuronales artificiales, reemplazada por el uso generalizado de neuronas sigmoidales, que suelen funcionar mejor en redes neuronales pequeñas. Hasta ya entrado el siglo XXI, se mantuvo la superstición de que había que evitar a toda costa funciones de activación que no fuesen diferenciables en algún punto.

En realidad, no se dispone de una teoría completa acerca de cuándo son preferibles las unidades lineales rectificadas, ni por qué exactamente. No obstante, evitar los efectos perniciosos de la saturación de las neuronas sigmoidales suele resultar beneficioso, tanto para problemas prácticos concretos, p.ej. reconocimiento de imágenes,²⁶⁰ como para modelos alternativos de redes neuronales artificiales, p.ej. máquinas de Boltzmann.²⁶¹

Otras funciones de activación alternativas

A veces ni siquiera resulta necesario utilizar funciones no lineales de activación. Cuando se prescinde de una función de activación no lineal, se hace en parte de la red, no en toda. De esta forma, un uso razonable de neuronas lineales permite factorizar las matrices de pesos, lo que nos puede servir para reducir el número de parámetros de la red. Una capa de n entradas y m salidas puede sustituirse por dos capas, una primera capa lineal de n entradas y p salidas intermedias, seguida de una capa con las mismas características que la original, pero de p entradas y m salidas. En el primer caso, la capa tendrá $O(nm)$ parámetros. En el segundo, $O((n+m)p)$ parámetros, que puede ser mucho menor que $O(nm)$ eligiendo adecuadamente el valor p . En algunas aplicaciones, como el procesamiento de imágenes con redes convolutivas, es habitual el uso de capas lineales de este tipo.

Otras funciones de activación que podrían utilizarse en una red neuronal artificial, al margen de su plausibilidad biológica, son las siguientes:

- Funciones de base radial RBF [*Radial Basis Function*]: Básicamente, evalúan la similitud entre el vector de entradas y su vector de paráme-

²⁶⁰ Alex Krizhevsky, Ilya Sutskever, y Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. En *NIPS'2012 Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL <https://goo.gl/Ddt8Jb>

²⁶¹ Vinod Nair y Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. En Johannes Fürnkranz y Thorsten Joachims, editores, *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814. Omnipress, 2010. ISBN 978-1-60558-907-7. URL <https://www.cs.toronto.edu/~hinton/absps/reluICML.pdf>

tros. Sin embargo, al saturarse para la mayor parte de sus valores de entrada, resultan difíciles de entrenar utilizando gradiente descendente y *backpropagation*.

- La función *softplus*: Se trata de una versión suavizada, continua y derivable, de las unidades ReLU. Pese a sus mejores propiedades desde el punto de vista teórico, empíricamente suele ofrecer peores resultados que el uso de unidades ReLU, que además resultan más eficientes al prescindir de funciones trascendentales en su evaluación.
- La tangente hiperbólica estricta [*hard tanh*], una versión linealizada de la tangente hiperbólica que no es más que una función lineal con saturación: $f_{\text{hard tanh}}(z) = \max\{-1, \min\{1, z\}\}$.
- Sorprendentemente, funciones trigonométricas como el coseno, $f_{\cos}(z) = \cos(z)$, pueden funcionar bien en situaciones puntuales. Por ejemplo, usando la base de datos MNIST, esta función de activación consigue una tasa de error inferior al 1%, competitiva con la que se puede conseguir utilizando otras funciones de activación más convencionales.

El abanico de alternativas posibles es amplio, ya que el factor crítico es la arquitectura multicapa de la red más que el tipo de no linealidad que se introduzca en sus unidades ocultas. Se pueden probar múltiples tipos de funciones de activación en las capas ocultas de una red y muchas de ellas pueden funcionar correctamente. Siempre, claro está, que sean diferenciables o, al menos, tengan definidas sus derivadas laterales en los puntos en los que no sean diferenciables.

Muchas funciones de activación distintas a las que hemos descrito pueden ser igual de válidas que sus alternativas más populares, aunque su uso no se haya generalizado. Hay tantas posibilidades que ofrecen resultados comparables a los ya conocidos, que no se suelen considerar interesantes desde el punto de vista formal. En la investigación de nuevas técnicas, no obstante, puede ser recomendable probar distintos tipos de funciones de activación. Aunque los resultados no suelan ser publicables (para eso habría que demostrar que proporcionan mejoras significativas, algo poco probable), siempre nos pueden ayudar a analizar el comportamiento de los algoritmos de entrenamiento de redes neuronales bajo diferentes situaciones, para delimitar su ámbito de aplicabilidad.

Puede que en el futuro se generalice el uso de funciones de activación de tipos hasta ahora desconocidos, ya que tal vez resulten útiles en la resolución de problemas prácticos concretos. Al fin y al cabo, no existe ninguna teoría sólida que nos indique cómo han de escogerse las funciones de activación de las neuronas de una red multicapa. Mientras tal teoría siga sin existir, siempre quedará la posibilidad de realizar nuevos descubrimientos y disfrutar de algunos momentos “¡Eureka!”, aunque sea en privado.

Modos de entrenamiento

La esencia del entrenamiento de redes neuronales consiste en utilizar una señal de error, analizar cómo fluctúa ese error en función de los parámetros de la red y ajustar dichos parámetros con la intención de reducir el error observado, en dirección del gradiente descendente:

$$\Delta w = -\eta \nabla E$$

La expresión anterior, en notación vectorial, se traduce en el cálculo de derivadas parciales del error con respecto a cada uno de los parámetros de la red:

$$\Delta w_{ij}^k = -\eta \frac{\partial E}{\partial w_{ij}^k}$$

Una de las decisiones que hemos de tomar a la hora de entrenar una red neuronal es con qué frecuencia ajustaremos los pesos durante el entrenamiento de la red o, dicho de otro modo, cómo estimamos el error de la red para ajustar sus parámetros. Tenemos tres opciones:

- *Entrenamiento por lotes [batch learning]*: Se ajustan los pesos una vez por cada época. Se recorre el conjunto de entrenamiento completo, acumulando el error para cada uno de los ejemplos de entrenamiento y, al final del recorrido, se realiza una actualización de los pesos de la red.
- *Entrenamiento online [online learning]*: Se ajustan los pesos de la red cada vez que le mostramos a la red un ejemplo de entrenamiento.
- *Entrenamiento por minilotes [mini-batch learning]*: Se ajustan los pesos a partir del error estimado para una pequeña muestra del conjunto de entrenamiento.

En principio, la estimación del error será más fiable si utilizamos todos los datos disponibles (aprendizaje por lotes) que si empleamos sólo un subconjunto del conjunto de entrenamiento (aprendizaje por minilotes y *online*), lo que nos permitirá obtener una estimación más realista del gradiente del error. Cuando se utiliza sólo una muestra del conjunto de entrenamiento, la muestra se suele seleccionar de manera estocástica (no determinística): la actualización de los parámetros de la red se realiza utilizando una estimación estocástica del gradiente del error, motivo por el que se suele denominar gradiente descendente estocástico [*SGD: Stochastic Gradient Descent*] a los modos de entrenamiento *online* y por minilotes.

Elegir entre el gradiente descendente (determinista) del entrenamiento por lotes o el gradiente descendente estocástico del aprendizaje *online* implica ciertos compromisos [*trade-offs*], que se intentan solucionar con la solución intermedia proporcionada por el entrenamiento por minilotes.

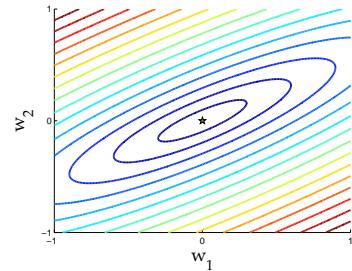


Figura 121: Contorno de la superficie de error asociada a una neurona lineal. La dirección del gradiente es perpendicular a las líneas del diagrama, que podemos interpretar como los mapas de isobares del pronóstico del tiempo. Aunque en las neuronas no lineales la superficie de error no sea cuadrática, localmente nos vale como aproximación a la hora de interpretar los distintos modos de entrenamiento de una red neuronal artificial.

Entrenamiento por lotes

En el entrenamiento o aprendizaje por lotes [*batch learning*], también conocido como gradiente descendente determinístico o de lote completo [*full-batch gradient descent*], se utilizan todos los ejemplos del conjunto de entrenamiento para estimar el gradiente del error. Esto es, se evalúa el gradiente del error para cada ejemplo individual del conjunto de entrenamiento y se promedian los resultados individuales para obtener una estimación del gradiente lo más parecida posible al gradiente real de la función de error:

$$\nabla E(w) = \frac{1}{m} \sum_{i=1}^m \nabla E(x_i, y_i; w)$$

donde (x_i, y_i) es un ejemplo de entrenamiento y $\nabla E(x_i, y_i; w)$ es el gradiente de la función de error evaluado para el ejemplo de entrenamiento (x_i, y_i) dados los parámetros w , correspondientes a los pesos de la red.

Aunque para las neuronas no lineales la superficie de error no es cuadrática, podemos utilizar una aproximación local de tipo cuadrático para entender los problemas que pueden aparecer durante el entrenamiento de la red. Incluso cuando utilizamos el conjunto de entrenamiento completo para estimar el gradiente del error, si existe cierta correlación entre los parámetros del error, las secciones transversales de la superficie de error serán elípticas, con elipses más alargadas cuanto mayor correlación exista. Esto hace que el gradiente de la función de error, que es perpendicular a esas secciones transversales, no siempre apunte en dirección al mínimo real de la función de error, algo que sólo sucedería cuando la sección transversal fuese una circunferencia perfecta. Es más, cuando estamos cerca de los extremos de la elipse correspondiente a la sección transversal de la superficie de error, la magnitud del gradiente es mucho mayor en la dirección perpendicular a la dirección hacia la que nos gustaría movernos, la que apunta al mínimo. El gradiente es elevado en una dirección en la que nos gustaría recorrer una distancia pequeña pero reducido en la dirección en la que realmente estamos interesados. Nos gustaría dar un salto grande en dirección al mínimo pero la forma de la superficie de error hace que el gradiente nos lleve en una dirección equivocada. Esto puede obligarnos a utilizar tasas de aprendizaje ridículamente pequeñas ($\eta << 1$) para asegurarnos de que vamos avanzando poco a poco en la dirección correcta, sin pasarnos del mínimo.

¿Qué es lo que realmente nos gustaría conseguir? Movernos más rápido en direcciones con gradientes pequeños pero consistentes, más despacio en direcciones con gradientes grandes pero inconsistentes. Más o menos, lo que se consigue incorporando momentos en la actualización de los pesos de la red:

$$\Delta w(t) = -\eta \nabla E + \alpha \Delta w(t-1)$$

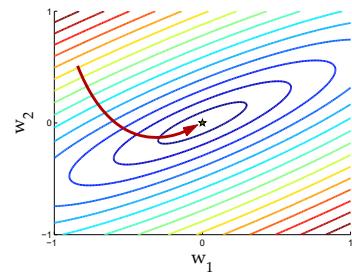


Figura 122: Aprendizaje por lotes: La estimación del gradiente utilizando el conjunto de entrenamiento completo permite avanzar en la dirección del gradiente real, aunque éste no siempre apuntará directamente al mínimo de la función de coste.

Otro inconveniente del entrenamiento por lotes es que sólo se realiza una actualización de los pesos en cada época de entrenamiento. Para que el cálculo del gradiente del error sea fiable, se necesitará un conjunto de entrenamiento grande. El coste computacional del cálculo del gradiente será proporcional al tamaño del conjunto de entrenamiento, $O(m)$. Si nuestro conjunto incluye millones de ejemplos, como suele ser habitual en *big data*, el coste de una simple iteración del algoritmo puede resultar prohibitivo. Dado que, además, se necesitarán múltiples iteraciones para conseguir que el algoritmo converja, el coste computacional del entrenamiento de la red puede ser demasiado elevado.

El entrenamiento por lotes de una red neuronal suele ser muy lento y requerir demasiado tiempo. Ahora bien, si nos fijamos en nuestro algoritmo de entrenamiento, lo que estamos haciendo al calcular el gradiente es intentar descubrir una dirección en la que podamos mover los valores de los parámetros de la red con la intención de disminuir su error. En realidad, no necesitamos el gradiente real de la función de error (que, además, puede no ser demasiado buen guía en dirección al mínimo de la función de error). Nos basta realizar una estimación aproximada del gradiente para guiarnos en nuestro proceso iterativo de optimización. Dicha aproximación se puede calcular utilizando sólo un pequeño conjunto de muestras, sin necesidad de recurrir al conjunto de entrenamiento completo cada vez que queramos darle un pequeño empujón a los parámetros de nuestra red. Es lo que hace el gradiente descendente estocástico.

El gradiente descendente estocástico, en el que la estimación del gradiente se hace de forma aproximada, seleccionando al azar una pequeña muestra del conjunto de entrenamiento, nos permitirá conseguir un método escalable para entrenar modelos no lineales a partir de conjuntos de entrenamiento enormes.

Entrenamiento online

Llevando al extremo la idea del gradiente descendente estocástico, se obtiene el aprendizaje *online*. La estimación del gradiente del error se realiza directamente a partir del error observado para un único ejemplo de entrenamiento:

$$\nabla E(w) \approx \nabla E(x_k, y_k; w)$$

donde (x_k, y_k) es un ejemplo de entrenamiento concreto. Como sólo se emplea un ejemplo para cada actualización de los parámetros de la red, el modo de aprendizaje *online* en ocasiones recibe el nombre de aprendizaje incremental.

La estimación del gradiente, obviamente, dependerá mucho del ejemplo concreto que se escoja en cada momento, por lo que nos moveremos haciendo zigzag sobre la superficie de error.

¿En qué orden se le presentan los ejemplos de entrenamiento a la red?

Antes de la popularización del *deep learning*, la alternativa más popular para construir modelos no lineales consistía en recurrir a las máquinas de vectores de soporte, SVM, que combinan el truco del *kernel* con un modelo lineal. Sin embargo, esto suele requerir la construcción de una matriz de Gram $G_{ij} = k(x_i, x_j)$ de tamaño $m \times m$. Su coste computacional, de orden cuadrático, $O(m^2)$, hace que este tipo de técnicas no resulte realmente escalable.

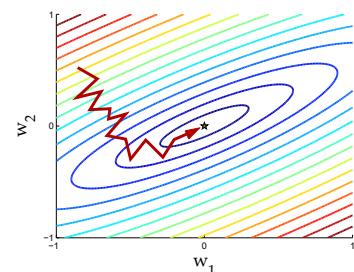


Figura 123: Aprendizaje *online*: La estimación estocástica del gradiente introduce ruido en el proceso de optimización basado en el gradiente descendente, por lo que no todas las actualizaciones de los pesos se realizan en la dirección del gradiente real de la función de error.

Suele ser preferible hacerlo aleatoriamente. LeCun recomienda, como regla general, intentar maximizar la información que un ejemplo de entrenamiento proporciona a la red. Para ello, se pueden utilizar dos sencillas heurísticas: seleccionar el ejemplo que resulte en un error mayor o seleccionar un ejemplo que sea los más diferente posible a los ejemplos utilizados antes que él.

En problemas de clasificación, conviene evitar mostrarle a la red todos los ejemplos de una clase antes de pasar a los de la clase siguiente. Igual que nos puede suceder a nosotros cuando caemos en la rutina, la red podría llegar olvidar lo aprendido anteriormente. En nuestro caso sería aún peor, ya que nuestra motivación se vería probablemente afectada en un trabajo rutinario, con lo que podríamos deprimirnos y llegar incluso a perder el interés por aprender nuevas cosas. Afortunadamente, una máquina carece de estado de ánimo, por lo que no se quejará aunque estemos haciendo mal las cosas. Esto resulta un arma de doble filo, ya que podemos no darnos cuenta de nuestro error a pesar de que el rendimiento de la red neuronal se vea seriamente afectado.

Cuando se utiliza aprendizaje *online*, una estrategia habitual consiste en barajar [*shuffle*] el conjunto de entrenamiento, de forma que en cada época del entrenamiento los ejemplos se le muestren a la red en un orden diferente. En ocasiones, la reordenación de los datos se realiza explícitamente para evitar que ejemplos sucesivos pertenezcan a una misma clase, aunque esto no suele ser crítico si las distintas clases están balanceadas y disponemos de un número de ejemplos similar para todas las clases de nuestro problema. Se puede argumentar formalmente que el uso de distintas permutaciones del conjunto de datos en cada época de entrenamiento [*almost-cyclic learning*] es preferible a utilizar siempre la misma permutación [*cyclic-learning*] o emplear un aprendizaje *online* completamente aleatorio, en el que cada muestra se escoja aleatoriamente de forma independiente, por lo que dos presentaciones de un mismo ejemplo de entrenamiento podrían estar separadas por más de una época.²⁶²

Mientras la tasa de aprendizaje siga siendo lo suficientemente pequeña como para que no nos pasemos del mínimo, se observa una relación lineal entre la tasa de aprendizaje η y el número de épocas necesario para el aprendizaje. La tasa de aprendizaje utilizada en el aprendizaje *online* puede ser mayor que la empleada en el aprendizaje por lotes.

Si tenemos un conjunto de entrenamiento de 10,000 ejemplos, seleccionamos una tasa de aprendizaje $\eta = 0.1$ y el gradiente del error con respecto a un peso es del orden de ± 0.1 para cada ejemplo, el cambio acumulado para un peso en cada época del aprendizaje por lotes será del orden de $\pm 0.1 \times 0.1 \times 10000 \approx \pm 100$. Este cambio es demasiado elevado, por lo que puede generar oscilaciones y hacer que el algoritmo no converja. Sin embargo, cuando usamos la misma tasa de aprendizaje en el entrenamiento *online*, el cambio en los pesos será del orden de

Presentar los ejemplos que producen mayores errores con mayor frecuencia que los ejemplos que producen menores errores puede ayudar a agilizar el ajuste de los parámetros de la red durante su entrenamiento.

²⁶² Tom Heskes y Wim Wiegerinck. A theoretical comparison of batch-mode, on-line, cyclic, and almost-cyclic learning. *IEEE Transactions on Neural Networks*, 7(4):919–925, July 1996. ISSN 1045-9227. DOI: 10.1109/72.508935

$\pm 0.1 \times 0.1 \approx \pm 0.01$. Así pues, para un valor de la tasa de aprendizaje η_{batch} que permita la convergencia del algoritmo de entrenamiento por lotes, la versión equivalente del algoritmo de aprendizaje *online* podría utilizar una tasa de aprendizaje $\eta_{online} = m \eta_{batch}$, donde m es el tamaño del conjunto de entrenamiento. En la práctica, se recomienda $\eta_{online} = \sqrt{m} \eta_{batch}$.

El coste computacional de procesar cada ejemplo de entrenamiento es similar en aprendizaje *online* y aprendizaje por lotes. Tanto en uno como en otro, se necesita espacio suficiente para almacenar todos los pesos de la red, N_w . Dado un ejemplo de entrenamiento, la propagación de la señal de entrada para obtener la salida de la red requerirá N_w multiplicaciones. La propagación hacia atrás del error requerirá $2N_w$ multiplicaciones. Las actualizaciones de los pesos requerirán multiplicar el gradiente por la tasa de aprendizaje, N_w multiplicaciones adicionales, para cada ejemplo en el entrenamiento *online*, una vez por época en el entrenamiento por lotes. Esto es, se necesitan 4 multiplicaciones por peso y por ejemplo en el aprendizaje *online*, tres en el aprendizaje por lotes.

En el aprendizaje *online* se actualizan los parámetros de la red m veces durante cada época de entrenamiento, frente a la única actualización del entrenamiento por lotes. Esto suele hacer que el entrenamiento de redes usando el gradiente descendente estocástico y *backpropagation* sea mucho más rápido que el entrenamiento por lotes.²⁶³ La diferencia se acentúa cuando los conjuntos de entrenamiento son grandes e incluyen cierto grado de redundancia, lo que suele ser habitual en la práctica.

Así pues, el entrenamiento de redes neuronales basado en el gradiente descendente y *backpropagation* ofrece algunas ventajas con respecto al aprendizaje por lotes:

- El entrenamiento *online* o aprendizaje incremental suele ser órdenes de magnitud más rápido que el aprendizaje por lotes, siendo al menos tan preciso como este último (especialmente, para conjuntos de datos grandes). Su convergencia es más rápida al poder utilizar tasas de aprendizaje más elevadas de forma segura.
- Pese a utilizar una peor aproximación del gradiente real en el ajuste de los parámetros de la red, como se va recorriendo una trayectoria sobre la superficie de error durante cada época de entrenamiento, el aprendizaje *online* suele obtener mejores soluciones que el aprendizaje por lotes cuando limitamos la duración del entrenamiento de la red.
- La estrategia incremental utilizada por el aprendizaje *online* facilita que la red pueda adaptarse a cambios que se puedan producir en el conjunto de datos de entrenamiento. Esto puede resultar de interés en el procesamiento de *data streams*, para los que podemos ir actualizando nuestro modelo cada vez que nos llegan datos nuevos.

²⁶³ D.Randall Wilson y Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429 – 1451, 2003. ISSN 0893-6080. doi: 10.1016/S0893-6080(03)00138-2

Pese a las claras ventajas del aprendizaje *online*, también pueden existir motivos que justifiquen el uso del aprendizaje por lotes. Entre las posibles ventajas que ofrece el aprendizaje por lotes, se encuentran las siguientes:

- Condiciones de convergencia bien conocidas:
 - Cuanto se utiliza el error cuadrático, MSE, el aprendizaje por lotes garantiza la convergencia del algoritmo de aprendizaje siempre que la tasa de aprendizaje $\eta < 2$.
 - Cuando se utiliza el error en valor absoluto, MAE, si el conjunto de entrenamiento es lo suficientemente grande, el algoritmo de aprendizaje por lotes converge más rápidamente que el algoritmo de aprendizaje *online* cuando $\eta < 1.2785$.
- Muchas técnicas de optimización sólo funcionan cuando se utiliza aprendizaje por lotes y se realiza una estimación fiable del gradiente del error. Muchas técnicas matemáticas de optimización permiten acelerar el aprendizaje de los parámetros de una red multicapa, p.ej. gradientes conjugados o L-BFGS. No obstante, los métodos existentes de optimización no lineal, propuestos para otros tipos de problemas, pueden necesitar ciertas adaptaciones para que funcionen correctamente con redes neuronales artificiales.
- El entrenamiento por lotes permite realizar un análisis formal de las propiedades dinámicas del proceso de aprendizaje y de su convergencia, algo que no se puede hacer si utilizamos aprendizaje *online*. Este análisis teórico puede resultar de interés en determinadas aplicaciones, para las que sea necesario garantizar formalmente determinadas propiedades del modelo que luego se utiliza en un sistema real.

Algunos análisis teóricos demuestran que, usando funciones de activación sigmoidales, el entrenamiento *online* de redes neuronales multicapa siempre converge bajo ciertas condiciones, no demasiado exigentes.²⁶⁴ Usando el error cuadrático, el aprendizaje *online* converge si la varianza de las estimaciones del gradiente para cada ejemplo decrece exponencialmente con el número de épocas de entrenamiento. Con una tasa de aprendizaje adecuada η y cierto ritmo de disminución de la varianza de las estimaciones del gradiente, se puede demostrar que el aprendizaje incremental converge más rápidamente que el aprendizaje por lotes.

Existen estudios similares^{265,266} en los que se intentan analizar las propiedades formales del aprendizaje *online* frente a otros modos de entrenamiento, como el aprendizaje por lotes. No obstante, estos resultados teóricos son de una utilidad práctica, digamos, algo limitada. Además de resultar menos generales que los resultados equivalentes que se podrían derivar para el aprendizaje por lotes.

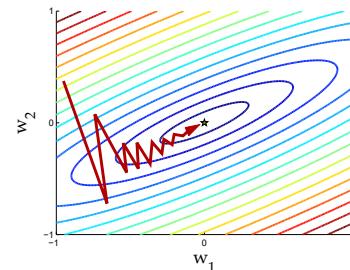


Figura 124: Aprendizaje por lotes cuando el algoritmo converge ($\eta < 2$).

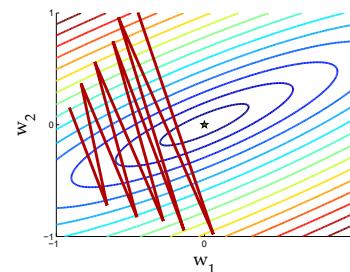


Figura 125: Aprendizaje por lotes cuando el algoritmo no converge debido al uso de una tasa de aprendizaje demasiado alta ($\eta > 2$).

²⁶⁴ Zong-Ben Xu, Rui Zhang, y Wen-Feng Jing. When Does Online BP Training Converge? *IEEE Transactions on Neural Networks*, 20(10):1529–1539, Oct 2009. ISSN 1045-9227. DOI: 10.1109/TNN.2009.2025946

²⁶⁵ Takéhiko Nakama. Theoretical analysis of batch and on-line training for gradient descent learning in neural networks. *Neurocomputing*, 73(1): 151 – 159, 2009. ISSN 0925-2312. DOI: 10.1016/j.neucom.2009.05.017

²⁶⁶ Rui Zhang, Zong-Ben Xu, Guang-Bin Huang, y Dianhui Wang. Global Convergence of Online BP Training with Dynamic Learning Rate. *IEEE Transactions on Neural Networks and Learning Systems*, 23(2):330–341, Feb 2012. ISSN 2162-237X. DOI: 10.1109/TNNLS.2011.2178315

Entrenamiento por minilotes

A medio camino entre el aprendizaje por lotes, que realiza una sola actualización de los pesos por época de entrenamiento, y el aprendizaje *online*, que actualiza los pesos cada vez que se le muestra un ejemplo a la red, se encuentra otra variante del gradiente descendente estocástico: el entrenamiento por minilotes. En el entrenamiento con minilotes, se calcula el gradiente del error a partir de una pequeña muestra de casos de entrenamiento elegidos al azar. Promediando sobre esta pequeña muestra, obtenemos una buena estimación del gradiente real de forma rápida:

$$\nabla E(w) = \frac{1}{m} \sum_{i=1}^m \nabla E(x_i, y_i; w) \approx \frac{1}{k} \sum_{j=1}^k \nabla E(x_{b(j)}, y_{b(j)}; w)$$

donde la primera sumatoria, usada en el aprendizaje por lotes, utiliza todos los ejemplos de entrenamiento (x_i, y_i) , pero la segunda sólo emplea los ejemplos $(x_{b(j)}, y_{b(j)})$ de un minilote seleccionado aleatoriamente del conjunto de entrenamiento.

La idea de la estimación del gradiente por minilotes consiste en que, si existe redundancia en el conjunto de entrenamiento, un fenómeno muy habitual, podríamos calcular el gradiente utilizando la primera mitad del conjunto de datos y obtendríamos un valor muy parecido al gradiente que se obtendría si utilizásemos la segunda mitad del conjunto de datos de entrenamiento. Dado que el gradiente estimado para las dos mitades del conjunto será similar, podríamos actualizar los pesos usando la estimación obtenida a partir de la primera mitad y volver a obtener una estimación del gradiente, ya para los pesos actualizados, usando la segunda mitad. En una sola época iríamos realizando múltiples actualizaciones de los parámetros de la red. Cuando el conjunto de entrenamiento incluye datos redundantes, ni siquiera tenemos que utilizar la mitad del conjunto para realizar una estimación estadística del gradiente. Podemos considerar una pequeña muestra y, con ella, obtener una buena estimación del gradiente. Llegamos, de esta forma, al uso de minilotes. Pequeñas muestras del conjunto de datos de entrenamiento que nos permiten realizar una buena estimación del gradiente real del error de forma muy eficiente, lo que acelera el gradiente descendente y nos permite obtener un algoritmo de entrenamiento de redes neuronales más rápido que el entrenamiento por lotes. Los algoritmos de optimización basados en el gradiente descendente convergen mucho más rápido, en términos de requisitos computacionales, no de número de actualizaciones, si realizamos estimaciones aproximadas del gradiente en vez de calcular el gradiente exacto.

¿Cómo se decide el tamaño del minilote? El entrenamiento por minilotes consiste en seleccionar un pequeño número k de ejemplos del conjunto de entrenamiento, lo suficientemente grande como para esperar que la estimación estocástica del gradiente sea similar al gradiente que

En la práctica, el cálculo del gradiente se suele realizar omitiendo los factores $1/m$ (entrenamiento por lotes) y $1/k$ (entrenamiento por minilotes) que aparecen antes de la sumatoria de las contribuciones al gradiente de cada ejemplo de entrenamiento. Formalmente, no supone ninguna diferencia, ya que equivale simplemente a reescalar la tasa de aprendizaje η del algoritmo.

obtendríamos utilizando todo el conjunto de entrenamiento. En su caso extremo, el correspondiente al aprendizaje *online* ($k = 1$), la estimación del gradiente no será demasiado fiable y realizaremos un recorrido casi aleatorio sobre la superficie de error. El error estándar del gradiente estimado a partir de k ejemplos viene dado por σ/\sqrt{k} , donde σ es la desviación estándar de la estimación del gradiente para los distintos ejemplos de entrenamiento. La raíz cuadrada del denominador nos indica que la recompensa de utilizar más muestras en la estimación del gradiente no llega a ser lineal, lo que nos permitirá emplear minilotes de tamaño pequeño. Si comparamos una estimación realizada con 10 ejemplos frente a otra realizada con 1000 ejemplos, la segunda requerirá 100 veces más cálculos que la primera, pero sólo reduce el error de la estimación en un factor de 10.

El cálculo simultáneo del gradiente para los ejemplos de un minilote puede paralelizarse fácilmente. Utilizando operaciones con matrices en vez de un bucle que procese los ejemplos secuencialmente, uno a uno, la evaluación del gradiente del error para un minilote se puede hacer de forma muy eficiente con ayuda de una GPU. De hecho, es habitual ajustar el tamaño k del minilote en función de la capacidad de cálculo paralelo de la GPU de la que dispongamos, de forma que el tiempo de reloj necesario para realizar una actualización de los pesos de la red resulte prácticamente el mismo usando minilotes y sin usarlos. Un tamaño demasiado pequeño del minilote hace que no aprovechamos al máximo los beneficios que nos ofrece el hardware paralelo de una GPU. Demasiado grande y no estaremos actualizando los pesos con la frecuencia con la que podríamos hacerlo.

A la hora de elegir el tamaño adecuado de los minilotes, hay que considerar distintos factores para llegar a un punto de equilibrio en el que se maximice la velocidad del aprendizaje:

- El retorno sublineal, en la estimación del gradiente, cuanto mayor es el tamaño del minilote (σ/\sqrt{k}).
- La varianza es la estimación del gradiente, que será mayor cuanto menor sea el minilote. Esto nos obligará a utilizar tasas de aprendizaje más pequeñas para conservar la estabilidad del algoritmo de aprendizaje y el entrenamiento puede requerir más tiempo de ejecución.
- El efecto regularizador de los minilotes pequeños: Al añadir ruido al proceso de entrenamiento de la red, como en el aprendizaje *online*, lo que puede facilitar la convergencia del algoritmo.
- La infroutilización del hardware paralelo: Dada la arquitectura de las GPU, es habitual utilizar minilotes cuyo tamaño venga dado en potencias de 2, de 16 (2^4) a 256 (2^8).

El aprendizaje *online* y el aprendizaje con minilotes son dos formas diferentes del gradiente descendente estocástico. El uso de minilotes suele funcionar mejor que el aprendizaje *online*.

Si usamos aprendizaje *online*, $k = 1$, el uso de un único ejemplo de entrenamiento introducirá errores significativos en nuestra estimación del gradiente. En realidad, no necesitamos una estimación demasiado precisa, sólo una que nos permita movernos en una dirección adecuada, que nos facilite ir reduciendo el error de la red. Es como si nos movemos por el campo guiados por la posición del Sol. No necesitamos una brújula especialmente precisa, basta con comprobar con frecuencia en qué dirección nos estamos moviendo. Si nuestra estimación es, en media, más o menos correcta, acabaremos llegando a nuestro destino.

Para entrenar redes neuronales grandes a partir de conjuntos de entrenamiento enormes, que suelen ser muy redundantes, casi siempre es preferible utilizar minilotes. Aun cuando algunos métodos de optimización pueden requerir que el tamaño de los minilotes sea elevado, siempre resultan más eficientes computacionalmente que el uso de aprendizaje por lotes. Si no disponemos de hardware paralelo, el aprendizaje *online* suele ser más rápido.

La característica más importante del gradiente descendente estocástico (tanto *online* como con minilotes) es que los recursos computacionales necesarios por actualización de los parámetros de la red no crecen con el tamaño del conjunto de entrenamiento, lo que contribuye a la escalabilidad del algoritmo. El gradiente descendente estocástico, SGD, funciona para conjuntos de datos enormes, cuyo procesamiento sería demasiado ineficiente utilizando aprendizaje por lotes. De hecho, para conjuntos de entrenamiento gigantescos, puede que SGD converja incluso antes de haber procesado el conjunto de entrenamiento completo (¡en menos de una época!).

Durante el entrenamiento de la red, para elegir los minilotes, lo ideal sería realizar un muestreo aleatorio, con reemplazo, del conjunto de entrenamiento completo, de forma que se consiga una estimación no sesgada del gradiente del error. En teoría, podría ayudarnos a acelerar el aprendizaje si elegimos los ejemplos que proporcionan mayor información para el entrenamiento de la red. De esta forma, presentando los ejemplos que producen mayores errores con mayor frecuencia que los ejemplos que producen menos errores, podríamos acelerar el proceso de ajuste de los parámetros de la red. No obstante, esta estrategia podría no ser viable en la práctica, pero como mínimo debemos barajar los datos de entrenamiento, como en el aprendizaje *online*, para que los ejemplos incluidos en un minilote no estén correlacionados. En problemas de clasificación, se suelen balancear los ejemplos de las distintas clases de nuestro problema en cada minilote, intentado que todos los ejemplos de un minilote no pertenezcan a la misma clase, lo que sesgaría la estimación

del gradiente del error.

Desde el punto de vista formal, la capacidad del gradiente descendente estocástico de progresar rápidamente evaluando el gradiente a partir de sólo algunos ejemplos compensa su convergencia asintóticamente más lenta que si usásemos gradiente descendente por lotes.²⁶⁷ De hecho, podríamos intentar aprovechar las ventajas teóricas de ambos, aumentando progresivamente el tamaño del minilote conforme avanza el entrenamiento de la red.

En la práctica, se suele comenzar escogiendo valores aceptables, aunque no necesariamente óptimos, para otros hiperparámetros del algoritmo de aprendizaje. Con esos valores, se prueban distintos tamaños de minilote, con el objetivo de seleccionar el que proporciona mayores mejoras en la velocidad de entrenamiento de la red. Una vez seleccionado el tamaño del minilote, se procede a optimizar el resto de hiperparámetros del entrenamiento de la red, para lo que se pueden adoptar estrategias heurísticas complementarias que ayuden a acelerar su convergencia.²⁶⁸

El conjunto de entrenamiento

Como sucede con cualquier otra técnica de aprendizaje automático, el rendimiento de una red neuronal depende de forma crítica del conjunto de datos de entrenamiento que se haya utilizado para ajustar sus parámetros. Es fundamental que el conjunto de entrenamiento sea completo y se parezca a los datos con los que realmente se encontrará la red en la práctica, una vez entrenada y puesta en marcha en un entorno real.

En muchas ocasiones, las diferencias que se pueden conseguir ajustando los múltiples parámetros de un algoritmo de aprendizaje automático no son significativas en comparación con las mejoras que se pueden llegar a lograr si invertimos algo más de tiempo y esfuerzo en la recopilación y preparación de un conjunto de datos de entrenamiento adecuado para el problema que pretendamos resolver.

Preprocesamiento de los datos de entrenamiento

Normalmente, se recomienda normalizar los datos del conjunto de entrenamiento. La normalización de los datos de entrada se puede realizar empleando *z-scores* (variables tipificadas): $z = (x - \mu)/\sigma$, donde x es el valor original de la variable, μ es su media en el conjunto de entrenamiento y σ su desviación. De esta forma, cada variable del conjunto de entrenamiento pasa a tener media 0 y varianza 1. La normalización es, pues, una forma de compensar las diferencias que pueden existir entre las diferentes variables de entrada.

Para facilitar el entrenamiento de la red, suele ser recomendable que el conjunto de entrenamiento cumpla las siguientes propiedades:

²⁶⁷ Léon Bottou. Online Algorithms and Stochastic Approximations. En David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. ISBN 0521652634. URL <http://leon.bottou.org/publications/pdf/online-1998.pdf>. Revisado en junio de 2017

²⁶⁸ Mu Li, Tong Zhang, Yuqiang Chen, y Alexander J. Smola. Efficient mini-batch training for stochastic optimization. En *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 661–670, 2014b. ISBN 978-1-4503-2956-9. DOI: 10.1145/2623330.2623612

- La media de cada variable de entrada en el conjunto de entrenamiento debería ser cercana a cero.
- La escala de las variables de entrada debería ajustarse para que sus varianzas sean similares.
- Si es posible, las variables de entrada deberían estar decorreladas (sin correlación).

¿Por qué es importante que los datos satisfagan esas propiedades? El desplazamiento de la media contribuye a simplificar el entrenamiento del sesgo de las neuronas (al menos, en la primera capa de neuronas con pesos ajustables). El reescalado de los datos permite que los pesos de las neuronas de la capa de entrada no tengan que ajustar su escala en función de la variable concreta que reciben como entrada, de forma que sus escalas serán similares y convergerán al mismo ritmo. La decorrelación de los datos contribuye a que las secciones transversales de la superficie de error (elípticas en el caso del error cuadrático) sean menos alargadas, lo que facilita que la estimación del gradiente apunte correctamente en dirección al mínimo de la función de error.

El simple uso de *z-scores*, que involucra un desplazamiento de la media de cada variable y un escalado para normalizar su varianza, consigue las dos primeras propiedades. Para conseguir que las variables de entrada no estén correlacionadas, hace falta recurrir a técnicas un poco más sofisticadas. Aun sin recurrir a ellas, el simple hecho de hacer que la media de las variables de entrada sea cero puede suponer una gran ayuda a la hora de entrenar una red neuronal usando gradiente descendente. También suele ayudar que cada componente del vector de entrada tenga varianza 1 sobre el conjunto de entrenamiento completo, por lo que el uso de *z-scores* es más que recomendable.

En realidad, bastaría con preprocesar los datos de forma que cada variable de entrada tuviese una media pequeña en relación con su desviación estándar. Imaginemos qué sucedería si todas variables de entrada toman siempre valores positivos. Entonces, los pesos sinápticos de la primera capa oculta variarán todos siempre en el mismo sentido. O bien aumentarán todos, o bien disminuirán todos. El ajuste de los pesos sólo puede realizarse zigzagueando de un lado a otro de la superficie de error, por lo que la convergencia del algoritmo puede resultar extremadamente lenta. Un argumento similar nos hizo recomendar la tangente hiperbólica frente a la función logística como función de activación para las neuronas sigmoidales.

Para acelerar el proceso de aprendizaje, también se recomienda que las variables estén decorreladas, lo que se puede conseguir utilizando análisis de componentes principales [PCA: *Principal Component Analysis*].

Como método de decorrelación de las variables de entrada, el análisis de componentes principales permite convertir una superficie de error

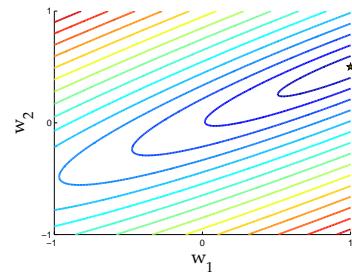


Figura 126: Superficie de error asociada a los datos originales.

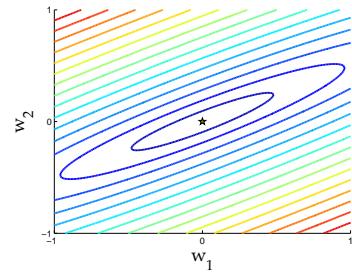


Figura 127: Superficie de error asociada a los datos una vez desplazados (variables con media 0).

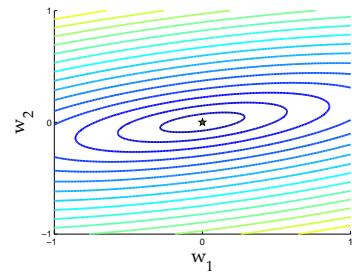


Figura 128: Superficie de error asociada a los datos con las variables tipificadas (*z-scores* con varianza 1).

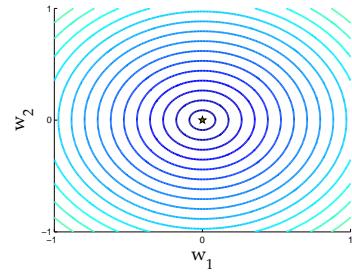


Figura 129: Superficie de error asociada a los datos decorrelados (p.ej. PCA).

elíptica en una circular, en la que el gradiente apunte directamente al mínimo (en superficies de error cuadráticas, que podemos utilizar como aproximación local de las superficies de error más complejas con las que nos encontraremos en la práctica). Para ello, tras eliminar la media y decorrelar las variables, se ecualiza la matriz de covarianza, dividiendo los componentes principales por las raíces cuadradas de sus respectivos *eigenvalues*.

El análisis de componentes principales, PCA, se puede utilizar no sólo para decorrelar las variables de entrada, sino como mecanismo de reducción de la dimensionalidad de los datos. Si eliminamos los componentes principales que tienen *eigenvalues* de menor valor, nos quedamos con aquellos componentes que explican la mayor parte de la variación observada en los datos de entrenamiento. Además de ser útil como mecanismo de visualización o de compresión de datos, la extracción de características usando PCA nos puede ayudar a reducir significativamente la dimensionalidad de los datos de entrada y, de paso, el número de parámetros necesarios para la red neuronal que utilicemos para procesar esos datos. Por ejemplo, dado un conjunto de imágenes para construir un sistema de reconocimiento facial, podríamos transformar los píxeles de las imágenes correspondientes a las caras en vectores de características asociados a sus componentes principales. Si partimos de pequeñas imágenes de 64×64 píxeles (vectores de datos de entrada de 4096 componentes), igual podríamos quedarnos sólo con 64 componentes principales de esas imágenes y construir un sistema usando sólo 64 componentes, con una eficacia similar al sistema que utilizase los 4096 componentes originales pero mucho más eficiente desde el punto de vista computacional.

En resumen, el preprocessamiento recomendado de los datos de entrada para acelerar el entrenamiento de una red neuronal artificial incluye tres etapas:²⁶⁹

- Anulación de medias, sustrayendo a cada valor de una variable su media en el conjunto de entrenamiento completo.
- Decorrelación de las variables de entrada, por medio del análisis de componentes principales, PCA.
- Ecualización de las covarianzas: Las variables decorreladas se escalan para que sus covarianzas sean aproximadamente iguales, lo que permite que los diferentes pesos de la red aprendan aproximadamente a la misma velocidad.

Cuando este preprocessamiento se combina con el uso de la tangente hiperbólica, se consigue que las salidas de las neuronas ocultas de una red multicapa tengan también media cero y varianza uno, debido a que

²⁶⁹ Yann LeCun. Efficient Learning and Second-Order Methods. *NIPS 1993 Tutorial*, 1993. URL <https://goo.gl/njHrMC>

la ganancia de la función de activación sigmoidal será también cercana a uno en su rango útil.²⁷⁰

Desde un punto de vista más teórico, se puede relacionar la velocidad de convergencia de algoritmos iterativos de optimización basados en el gradiente descendente con la naturaleza del conjunto de datos de entrenamiento. En ocasiones, un algoritmo de este tipo puede requerir un número de iteraciones proporcional a la dimensionalidad de los datos de entrada.

En análisis numérico, el número de condición [*condition number*] de una función con respecto a uno de sus argumentos mide cuánto puede cambiar su valor de salida dado un pequeño cambio en su entrada.

Por ejemplo, dado un sistema de ecuaciones lineales $Ax = b$, el número de condición de la matriz A nos sirve para acotar cómo de imprecisa será la aproximación de la solución x cuando se calcule la inversa A^{-1} .

Una matriz A se dice bien condicionada si su número de condición κ_A es cercano a 1 y se dice mal condicionada si κ_A es significativamente mayor que 1 [*poor/ill conditioning*]. Su número de condición se puede obtener como $\kappa_A = |\lambda_{\max}|/|\lambda_{\min}|$, la proporción entre su mayor *eigenvalue* λ_{\max} y su menor *eigenvalue* λ_{\min} , motivo por el que también se denomina *eigenvalue spread*.

Al realizar un análisis de la sensibilidad [*sensitivity analysis*] de los algoritmos de entrenamiento de redes neuronales, se puede relacionar su velocidad de convergencia con la matriz X de datos de entrenamiento: la convergencia del algoritmo es sensible al número de condición κ_X asociado a la matriz de datos de entrenamiento. El gradiente descendente para ajustar los parámetros w de funciones cuadráticas convexas verifica la siguiente condición:²⁷¹

$$f(w_{t+1}) - f(w^*) \leq \left[\frac{\kappa_X - 1}{\kappa_X + 1} \right]^2 (f(w_t) - f(w^*))$$

donde se puede observar que la convergencia del algoritmo depende de un parámetro ρ^2 que se define a partir del número de condición de la matriz de entrenamiento: $\rho = (\kappa_X - 1)/(\kappa_X + 1)$. Cuanto mayor sea este último, más cerca de 1 estará el parámetro ρ y más lenta será la convergencia del gradiente descendente.

Las evidencias empíricas y los indicios formales apuntan, por tanto, a favor del uso de técnicas de reducción de la dimensionalidad de los datos.

No obstante, aunque se utilizan muy a menudo con diversas técnicas de aprendizaje automático, están parcialmente mal vistas desde un punto de vista filosófico en *deep learning*, donde se supone que pretendemos que la red sea capaz de extraer información útil a partir de los datos en bruto, en forma de jerarquía de características aprendidas de un modo completamente automático.

²⁷⁰ Grégoire Montavon, Genevieve B. Orr, y Klaus-Robert Müller, editores. *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700 of *Lecture Notes in Computer Science*. Springer, 2012. ISBN 364235288X. DOI: 10.1007/978-3-642-35289-8

²⁷¹ David G. Luenberger y Yinyu Ye. *Linear and Nonlinear Programming*. International Series in Operations Research & Management Science. Springer, 4th edition, 2016. ISBN 3319188410

El mal condicionamiento de la matriz de datos es intrínseco a ella, algo que ningún algoritmo de aprendizaje podrá evitar mientras trabajemos directamente con esa matriz.

Ampliación del conjunto de entrenamiento

Una forma sencilla de conseguir mejorar el rendimiento de una red neuronal artificial es ampliar el conjunto de entrenamiento añadiéndole ejemplos creados artificialmente. Si, en una aplicación concreta esperamos encontrarnos con ejemplos distorsionados de determinadas formas, podemos generar esas mismas distorsiones en los ejemplos de nuestro conjunto de entrenamiento, ampliando su tamaño para preparar a la red de cara a ese tipo de distorsiones. Formalmente, estamos dotando a la red de una propiedad muy deseable en la práctica: invarianza frente a modificaciones habituales en los datos de entrada.

En una aplicación de reconocimiento de voz, por ejemplo, puede que esperemos que el sistema tenga que funcionar correctamente con fluctuaciones en el volumen, velocidad y tono de voz [*pitch*] de la persona que está hablando. Además, aunque los mecanismos de supresión de ruido han mejorado notablemente en los micrófonos que se utilizan en dispositivos actuales (o, para ser más precisos, en su software), es más que probable que el sistema deba funcionar con distintos ruidos de fondo e interferencias. Generando muestras sintéticas con esas características adicionales podemos contribuir a que una red neuronal funcione mejor en la práctica.

En aplicaciones que trabajan con imágenes, para las que se suelen utilizar redes convolutivas, también se pueden idear distintos tipos de transformaciones que contribuyan a ampliar el conjunto de datos de entrenamiento y mejorar el rendimiento de una red neuronal en la práctica. Las más habituales son rotaciones y cambios de escala. Por ejemplo, Patrice Simard y sus colaboradores de Microsoft Research emplearon esta estrategia sobre la base de datos MNIST.²⁷² Una sencilla red neuronal multicapa, como las que hemos visto, consigue una precisión del 98.4 % al clasificar dígitos manuscritos o, si lo prefiere, una tasa de error del 1.6 %. Añadiendo transformaciones afines al conjunto de entrenamiento (traslaciones, rotaciones e inclinaciones), la tasa de error de la red multicapa baja al 1.1 %. Si, además, se utilizan distorsiones elásticas, que físicamente podrían corresponder a los efectos causados por las oscilaciones de los músculos de la mano que está escribiendo, amortiguadas por la inercia, la tasa de error se reduce al 0.7 %. Con una arquitectura de red neuronal adecuada para este tipo de problema, una red convolutiva estándar para procesar imágenes, las tasas de error se reducen al 0.6 % usando transformaciones afines y al 0.4 % usando transformaciones elásticas, muy cerca de los mejores resultados que se pueden obtener usando un único modelo de clasificación.

Así pues, una vez más, conseguir un conjunto de entrenamiento tan grande como sea posible, aunque sea generando muestras sintéticas, permite a un modelo de aprendizaje convencional conseguir resultados

²⁷² Patrice Y. Simard, Dave Steinkraus, y John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. En *ICDAR'2003 Proceedings of the 7th International Conference on Document Analysis and Recognition*, pages 958–963, Aug 2003. DOI: 10.1109/ICDAR.2003.1227801

Se pueden conseguir resultados mejores empleando múltiples modelos; esto es, utilizando *ensembles*. Sin embargo, es difícil que mejoras puramente algorítmicas consigan mejorar los resultados que se obtienen ampliando el conjunto de entrenamiento, algo relativamente sencillo de hacer y que podemos aprovechar para mejorar el rendimiento de los modelos que entrenamos. En el caso de MNIST, una red con una topología más sofisticada que la usada por Simard, la red LeNet5 de Yann LeCun, obtenía una tasa de error del 0.8%: el doble de la que se obtiene al introducir transformaciones elásticas usando una red convolutiva más simple.

similares a los que se logran usando técnicas mucho más sofisticadas. En realidad, lo único que estamos haciendo es ampliar la experiencia de nuestra red durante su entrenamiento, exponiéndola al tipo de variaciones en los datos que se encontrará en el mundo real. Para ello, sólo tenemos que identificar cuáles son las operaciones que reflejan ese tipo de variaciones en la aplicación concreta para la que estemos diseñando la red neuronal. En el caso del reconocimiento de voz, puede ser ruido de fondo o cambios en la velocidad o en el tono de voz del hablante. En el caso de las imágenes, pueden ser distorsiones debidas a cambios en la posición de la cámara, en la iluminación de la escena o en el movimiento del objeto fotografiado [*motion blur*], como sucede en los radares de tráfico.

En ocasiones, en vez de ampliar el conjunto de entrenamiento creando muestras sintéticas, podemos ampliar el conjunto de entrenamiento obteniendo datos adicionales provenientes de distintas fuentes de datos.²⁷³ Por ejemplo, si trabajamos con textos en lenguaje natural, es fácil acceder a cantidades enormes de datos en la Web. El coste de adquisición de esos datos es, además, muy reducido, casi podríamos decir que gratis (nada es gratis del todo).

Los resultados obtenidos empíricamente demuestran que el uso de más datos en el entrenamiento de una red compensan las diferencias que pueden ofrecer los detalles particulares del algoritmo de aprendizaje automático utilizado. Obviamente, en la práctica, nos interesa conseguir los mejores resultados posibles, por lo que tendremos que llegar a un equilibrio entre el esfuerzo, en tiempo y dinero, que le dedicamos a la preparación de un conjunto de entrenamiento adecuado y el que le dedicamos a afinar el comportamiento del algoritmo de aprendizaje. Las recompensas más al alcance de nuestra mano puede que no siempre estén donde esperábamos encontrarlas.

¿Cómo sabemos si deberíamos dedicarle esfuerzo adicional a ampliar el conjunto de datos de entrenamiento? Básicamente, podemos analizar las tasas de error observadas en los conjuntos de entrenamiento y de prueba:

- Si el rendimiento de nuestra red neuronal sobre el conjunto de entrenamiento no es bueno, eso nos indica que el algoritmo de entrenamiento de la red ni siquiera está sacándole partido a los datos de los que ya disponemos. En principio, no parece que tengamos razones para recopilar más datos de entrenamiento para la red. Puede que, simplemente, debamos ampliar su capacidad, añadiéndole más capas y neuronas ocultas. También puede que necesitemos ajustar los hiperparámetros del algoritmo de entrenamiento. Tal vez estemos utilizando una tasa de aprendizaje demasiado elevada que impida la convergencia del algoritmo. Si al ampliar la capacidad de la red y ajustar sus hiperparámetros no conseguimos que su rendimiento mejore, el problema puede que

En el caso de las distorsiones debidas a diferentes tipos de ruido, a menudo se opta por añadir etapas de preprocesamiento de los datos de entrada para eliminar esos tipos de ruido, en vez de ampliar el conjunto de entrenamiento con ejemplos que incluyan ese ruido. La eliminación de ruido puede ser más eficiente que el entrenamiento de una red con un conjunto de datos ampliado.

²⁷³ Michele Banko y Eric Brill. Scaling to very very large corpora for natural language disambiguation. En *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL '01, pages 26–33, Stroudsburg, PA, USA, 2001. Association for Computational Linguistics. DOI: 10.3115/1073012.1073017

resida en la calidad de los datos de entrenamiento. Puede que tengan demasiado ruido e incluyan demasiados errores. O, simplemente, que no incorporen las características que realmente necesitaríamos para ser capaces de realizar predicciones acertadas. En escenarios así, tendremos que recopilar nuevos datos, más limpios (sin errores/ruido) y, quizá, con características adicionales.

- Si el rendimiento de nuestra red neuronal es mucho peor en el conjunto de prueba que en el conjunto de entrenamiento, recopilar más datos puede ser la estrategia más adecuada. Todo depende, claro está, del coste que eso suponga. En el caso particular de las redes neuronales, añadir una pequeña fracción al número total de ejemplos de entrenamiento puede no tener un impacto notable en el rendimiento del modelo entrenado. Normalmente se va modificando el tamaño del conjunto de entrenamiento usando una escala logarítmica; por ejemplo, doblando el número de ejemplos en experimentos consecutivos. Si el coste asociado a ampliar el conjunto de datos lo suficiente como para observar mejoras significativas en el rendimiento de la red es prohibitivo, ya sea por el coste de recopilar los datos en sí o el coste computacional asociado a entrenar una red usando muchos más datos, entonces no queda más remedio que buscar otras alternativas más económicas. Una de estas alternativas puede ser reducir el tamaño de la red o la capacidad efectiva del modelo utilizando técnicas de regularización, que estudiaremos en el próximo capítulo. Si estas técnicas no contribuyen a cerrar la brecha entre el rendimiento de la red en el conjunto de entrenamiento y en el de prueba, tal vez no quede más remedio que ampliar nuestro conjunto de entrenamiento.

Entrenamiento con adversario

Una de las ideas más populares en *deep learning* puede interpretarse como una forma de ampliar el conjunto de entrenamiento. En este caso, no diseñamos manualmente las transformaciones que aplicaremos sobre los datos originales del conjunto de entrenamiento ni buscamos fuentes de datos alternativas con las que completar los datos de los que ya disponemos. Simplemente, construimos una red neuronal que genere nuevos datos como los del conjunto de entrenamiento. Son los modelos de redes generativas con adversario: GAN [*Generative Adversarial Networks*].²⁷⁴

Propuestas en 2014 por Ian Goodfellow, entonces en la Universidad de Montréal, las redes GAN consisten en una pareja de redes: un generador y un discriminador. El objetivo del generador es sintetizar muestras que se parezcan lo más posible a las del conjunto de entrenamiento. El objetivo del discriminador es aprender a distinguir cuándo las muestras provienen del conjunto de datos real y cuándo son muestras sintéticas provenientes del generador. Formalmente, el generador crea ejemplos que

²⁷⁴ Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, y Yoshua Bengio. Generative adversarial nets. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger, editores, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014b. URL <http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>

parezcan provenir de la misma distribución que los datos del conjunto de entrenamiento (p_{data}). Esto es, aprende una distribución, p_{model} , que se parezca lo más posible a la distribución real de los datos, p_{data} . El discriminador es un clasificador binario, que intenta distinguir dos clases: real (proveniente de p_{data}) o falso (proveniente de p_{model}).

La idea básica de las redes GAN es configurar un juego entre dos jugadores: el generador y el discriminador. El generador se entrena para intentar confundir al discriminador, como si fuese un falsificador de dinero. El discriminador se entrena para intentar identificar al falsificador, actuando de policía. Para tener éxito, el falsificador debe ser capaz de crear dinero indistinguible del dinero auténtico; esto es, el generador debe aprender a crear ejemplos como los que aparecen en el conjunto de entrenamiento real. Cada jugador se puede representar por una función: la función G del generador genera muestras x a partir de variables latentes z (p.ej. ruido), la función D del discriminador genera una clasificación dada una muestra x . El juego se desarrolla en dos posibles escenarios:

- Se muestrean datos x del conjunto de entrenamiento y se utilizan como entrada del discriminador D . En este caso la salida del discriminador $D(x)$, que podemos interpretar como la probabilidad de que la muestra sea auténtica, debe ser cercana a 1.
- Dada una muestra z generada aleatoriamente de acuerdo a alguna distribución de probabilidad a priori, el generador G construye muestras falsas $G(z)$. A continuación, el discriminador intenta clasificar correctamente las muestras $G(z)$ como falsas. La salida correcta $D(G(z))$ debería ser próxima a cero. Sin embargo, aunque el discriminador intenta minimizar $D(G(z))$ para que se acerque a 0, el generador actúa para intentar que $D(G(z))$ se aproxime a 1.

El objetivo del discriminador D es estimar la probabilidad de que la muestra sea auténtica y no falsificada por el generador G . Bajo la suposición de que la mitad de las muestras son auténticas y la mitad son falsas, si los modelos del generador G y del discriminador D tienen la capacidad suficiente, el juego descrito alcanza su equilibrio Nash cuando las muestras $G(z)$ se toman de la distribución de los datos de entrenamiento ($p_{model} \equiv p_{data}$) y la salida del discriminador es $D(x) = 1/2$ para todo x .

Las funciones que describen al generador G y al discriminador D se eligen para que sean diferenciables con respecto tanto a sus entradas como a sus parámetros. De esta forma, tanto G como D pueden modelarse utilizando redes neuronales, que podemos entrenar usando gradiente descendente y *backpropagation*. El algoritmo de entrenamiento de una red GAN intercala el entrenamiento del generador G con el entrenamiento del discriminador D usando gradiente descendente estocástico:

- *Entrenamiento del generador:* Se muestrea un minilote z de muestras de acuerdo a $p_{model}(z)$ y se actualizan los parámetros w_G del generador G de acuerdo al gradiente de una función de coste J_G .
- *Entrenamiento del discriminador:* Se muestrea un minilote z de acuerdo a la distribución $p_{model}(z)$. A continuación, se muestrea un minilote x del conjunto de datos de entrenamiento (de la distribución real $p_{data}(x)$). Usando ambos minilotes, se actualizan los parámetros w_D del discriminador D de acuerdo al gradiente de una función de coste J_D .

El discriminador desea minimizar la función de coste $J_D(w_D, w_G)$ pero sólo puede controlar sus parámetros, w_D . De igual modo, el generador desea minimizar la función de coste $J_G(w_D, w_G)$, pero sólo puede actuar sobre sus propios parámetros, w_G . Como el coste de cada jugador depende no sólo de sus propios parámetros, sino de los de su adversario, sobre los que no tiene control directo, el entrenamiento de redes GAN se describe mejor como un juego de teoría de juegos en vez de como un problema de optimización. La solución del juego se obtiene cuando se llega a un equilibrio, denominado equilibrio de Nash, en el que ninguno de los jugadores tiene incentivos para modificar su forma de actuar.

La función de coste que se utiliza para el discriminador D es la habitual para problemas de clasificación: la entropía cruzada [*cross-entropy*]. Como el clasificador se entrena con dos minilotes que corresponden a cada una de las clases: el minilote z de muestras falsas provenientes del generador (clase negativa) y el minilote x proveniente del conjunto de datos de entrenamiento (clase positiva), la función de coste del discriminador es de la forma:

$$J_D = -\frac{1}{k} \sum_{i=1}^k \log(D(x_i)) - \frac{1}{k} \sum_{i=1}^k \log(1 - D(G(z_i)))$$

asumiendo minilotes de tamaño k .

En el caso del generador, se pueden utilizar diferentes funciones de coste:

- Minimax (como si de un juego de suma cero se tratase, cambiando el signo de la función de coste del discriminador):

$$J_G = -J_D$$

- Heurística (la entropía cruzada cambiando de objetivo, ya que el generador pretende conseguir muestras que parezcan auténticas y se entrena a partir de minilotes z):

$$J_G = -\frac{1}{k} \sum_{i=1}^k \log D(G(z_i))$$

En la propuesta original, se realizaban varias iteraciones de entrenamiento del discriminador para cada iteración de entrenamiento del generador, pero actualmente se suele aplicar un gradiente descendente ‘simultáneo’, intercalando una iteración de entrenamiento del generador con una iteración de entrenamiento del discriminador.

El nombre de John Forbes Nash le debería sonar a todo el que haya visto la película *Una mente maravillosa* [*A Beautiful Mind*, 2001], dirigida por Ron Howard y protagonizada por Russell Crowe. Película, por cierto, en la que se describe incorrectamente en qué consiste el equilibrio de Nash...

- Máxima verosimilitud (minimizando la divergencia Kullback–Leibler $D_{KL}(p_{data}||p_{model})$, una forma de medir el parecido entre dos distribuciones de probabilidad, p_{data} y p_{model}):

$$J_G = -\frac{1}{2} \frac{1}{k} \sum_{i=1}^k e^{\sigma^{-1}(D(G(z_i)))}$$

donde σ es la función logística.

En un estudio en el que colaboraron investigadores de Montréal, Nueva York, Google y Facebook, se hizo un descubrimiento inquietante.²⁷⁵ Incluso las redes neuronales entrenadas para conseguir un rendimiento equiparable al de un ser humano, si no mejor, tienen una tasa de error del 100 % sobre ejemplos diseñados a propósito para confundir a la red [*adversarial examples*].

En el caso de las redes generativas con adversario, GAN, el discriminador servía de adversario del generador, que de esa forma era capaz de conseguir ejemplos cada vez más parecidos a los reales. En el caso de los ejemplos diseñados por un adversario, ese adversario puede utilizar una estrategia similar para buscar una entrada x' muy parecida a una entrada real x de tal forma la salida de la red para x' sea completamente diferente a la salida para x . Un observador humano puede que ni siquiera se percate de que x' es diferente al x real, pero la red hará predicciones completamente distintas, lo que puede plantear serios problemas de seguridad en cualquier sistema que emplee internamente redes neuronales artificiales.

¿Cómo se pueden conseguir esos ejemplos capaces de confundir a una red neuronal sin que nosotros nos demos cuenta de ello? De la misma forma que la red se entrena usando el gradiente de una señal de error con respecto a los parámetros de la red, $\nabla_w J$, se puede usar el gradiente de la señal de error con respecto a los datos de entrada de la red, $\nabla_x J$, para maximizar el impacto de una perturbación imperceptible sobre su entrada. Diseñando ejemplos a medida, un adversario puede maximizar el error de la red... sin que nos demos cuenta.

Este hecho es realmente preocupante: las redes neuronales artificiales son vulnerables frente a ataques perpetrados por un adversario que diseñe ejemplos a medida. Además, las perturbaciones que generan errores en la red no son resultado del entrenamiento concreto de la red: las mismas perturbaciones pueden generar errores en redes con distintas arquitecturas que hayan sido entrenadas con conjuntos de entrenamiento diferentes. Esto es, ni siquiera es necesario que el atacante tenga acceso a los detalles particulares de la red utilizada en un sistema, sino que puede utilizar otra red diferente para diseñar sus ejemplos, con los que luego confundir a la red utilizada en el sistema real, realizando ataques de caja negra.

Los ejemplos diseñados por un adversario exponen auténticos puntos

²⁷⁵ Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, y Rob Fergus. Intriguing properties of neural networks. *ICLR'2014*, arXiv:1312.6199, 2014b. URL <http://arxiv.org/abs/1312.6199>

ciegos en el entrenamiento de una red neuronal. Esos puntos ciegos podemos intentar aprovecharlos para mejorar el entrenamiento de nuestras redes neuronales, ya sea para mejorar su rendimiento en el mundo real o para hacer frente a posibles ataques que pongan en jaque su seguridad. Esta es la idea que hay detrás del entrenamiento con adversario [*adversarial training*]: entrenar la red con ejemplos ligeramente modificados del conjunto de entrenamiento, tal como lo haría un posible adversario.²⁷⁶

La causa del punto ciego de las redes neuronales con respecto a ejemplos diseñados por un adversario se atribuía inicialmente, de forma especulativa, a la no linealidad de las redes neuronales multicapa, tal vez combinada con problemas de sobreaprendizaje. Actualmente, se cree que esas hipótesis son innecesarias: el comportamiento lineal de la red puede ser suficiente para explicar el fenómeno. Sí, ha leído bien, no se debe a la no linealidad de las redes neuronales, sino a su linealidad excesiva. Si cambiamos ligeramente cada entrada de la red con un pequeño valor ϵ , una función lineal en la que intervienen los pesos de la red, w , puede cambiar tanto como $\epsilon\|w\|_1$, que puede ser mucho cuando el número de parámetros de la red es elevado (como sucede con los vectores de pesos w de las redes utilizadas en *deep learning*).

El entrenamiento con adversario se puede realizar introduciendo un término adicional en la función de coste que se minimiza al entrenar la red, de forma que se fomente que la salida de la red sea más o menos constante en el vecindario de los ejemplos del conjunto de entrenamiento:

$$J_{adversarial}(x, y) = \alpha J(x, y) + (1 - \alpha)J(x + \epsilon \operatorname{sign}(\nabla_x J(x, y)), y)$$

donde $J(x, y)$ es la función de coste que utilicemos para entrenar la red dados los ejemplos (x, y) y α es un parámetro ajustable (p.ej. 0.5).

El entrenamiento con adversario evita el comportamiento excesivamente inestable de una red neuronal, el que ocasiona cambios bruscos en el entorno local de los ejemplos del conjunto de entrenamiento y acentúa su vulnerabilidad frente a los ataques de un adversario. Al utilizarlo, intentamos reducir el impacto de esos posibles ataques y, de paso, puede que mejoremos el comportamiento en la práctica de la red neuronal.

El entrenamiento con adversario también se puede emplear para resolver problemas de aprendizaje semisupervisado. Los ejemplos diseñados por el adversario, en este caso, no se generan utilizando la etiqueta auténtica, de la que no disponemos, sino una etiqueta proporcionada por un modelo entrenado a partir del conjunto de datos que sí tengamos etiquetado. Estos ejemplos se denominan ejemplos virtuales [*virtual adversarial examples*] y fomentan que el clasificador aprenda una función robusta frente a pequeños cambios en el subespacio donde residen los datos no etiquetados. La estrategia resultante recibe la denominación de entrenamiento con adversario virtual.²⁷⁷

Si queremos aplicar este tipo de técnicas en problemas de aprendizaje

²⁷⁶ Ian J. Goodfellow, Jonathon Shlens, y Christian Szegedy. Explaining and harnessing adversarial examples. *ICLR'2015*, arXiv:1412.6572, 2015a. URL <http://arxiv.org/abs/1412.6572>

²⁷⁷ Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, y Shin Ishii. Virtual Adversarial Training: a Regularization Method for Supervised and Semi-supervised Learning. *arXiv e-prints*, arXiv:1704.03976, 2016a. URL <http://arxiv.org/abs/1704.03976>

no supervisado, podemos recurrir al modelo con el que abrimos esta sección: las redes generativas con adversario. Dado que el generador GAN se entrena para confundir al discriminador, dicho generador puede terminar utilizándose como sustituto de una fuente de datos real. Si trabajamos con datos en forma de imagen, se pueden crear redes GAN con una arquitectura especializada para este tipo de datos: las redes convolutivas, que en el contexto de las redes GAN se denominan modelos DCGAN [*Deep Convolutional GAN*]. Los generadores DCGAN se han utilizado, por ejemplo, para sintetizar imágenes de cierta calidad visual. Las redes DCGAN pueden crear imágenes fotorrealistas que no existen en el conjunto de fotografías utilizadas como datos de entrenamiento, ya se trate de dormitorios, cambios intencionados en la pose de una persona o, incluso, cambios de sexo.²⁷⁸ Aún queda mucho camino por recorrer, pero sus primeros resultados resultan prometedores y, en ocasiones, sorprendentes.

Los modelos de redes con adversario, en sus distintas formas, gozan actualmente de cierta popularidad. Las redes generativas con adversario han atraído el interés de algunos investigadores, como un tipo particularmente interesante de modelo generativo que se puede entrenar con gradiente descendente y *backpropagation*.²⁷⁹ Para otros, el interés principal de los modelos con adversario reside en el análisis de las vulnerabilidades de los sistemas que internamente utilizan técnicas de aprendizaje automático, no sólo redes neuronales artificiales.²⁸⁰ Sin duda, la seguridad de tales sistemas es algo muy a tener en cuenta, dada la penetración actual de las técnicas de aprendizaje automático en múltiples ámbitos de nuestras vidas.

Desde el punto de vista formal, la ampliación del conjunto de datos de entrenamiento y el entrenamiento con adversario se pueden relacionar con técnicas de regularización que añaden términos adicionales a las funciones de coste con la que se entrena una red neuronal, con el objetivo de mejorar su rendimiento en la práctica reduciendo el sobreaprendizaje de la red:

- El método de propagación tangencial [*tangent prop*] intenta asegurar que la salida de la red, $f(x)$, sea localmente invariante frente a factores conocidos de variación.²⁸¹ Esto se consigue haciendo que el gradiente $\nabla_x f(x)$ sea ortogonal a los vectores v tangentes a la variedad [*manifold*] en el entorno de x donde se encuentran los datos de entrenamiento o, de forma equivalente, haciendo que las derivadas de f en x sean pequeñas en las direcciones dadas por los vectores v , para lo que se añade un término de regularización que penalice la función de coste con la que se entrena la red. La ampliación del conjunto de entrenamiento aplicando transformaciones concretas genera, de forma explícita, nuevos ejemplos que se añaden al conjunto de entrenamiento.

²⁷⁸ Alec Radford, Luke Metz, y Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *ICLR'2016*, arXiv:1511.06434, 2016. URL <http://arxiv.org/abs/1511.06434>

²⁷⁹ Ian J. Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. *arXiv e-prints*, arXiv:1701.00160, 2016. URL <http://arxiv.org/abs/1701.00160>

²⁸⁰ Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, y Ananthram Swami. Practical black-box attacks against machine learning. En *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 506–519, 2017. ISBN 978-1-4503-4944-4. DOI: 10.1145/3052973.3053009

²⁸¹ Patrice Simard, Bernard Victorri, Yann LeCun, y John Denker. Tangent Prop - A formalism for specifying selected invariances in an adaptive network. En J. E. Moody, S. J. Hanson, y R. P. Lippmann, editores, *NIPS'2001 Advances in Neural Information Processing Systems 4*, pages 895–903. Morgan-Kaufmann, 1992. URL <https://goo.gl/N3hQhR>

En ambos casos, el modelo aprendido resulta invariante frente al tipo de transformaciones especificado.

- El *backpropagation doble* [*double backprop*]^{282,283} regulariza el jacobiano de la red para que sea pequeño (la matriz jacobiana de una función matricial $f(x)$ contiene las derivadas parciales de cada una de sus salidas con respecto a cada una de sus entradas). El entrenamiento con adversario busca de forma explícita ejemplos similares a las entradas que podrían confundir a la red [*adversarial examples*] y entrena el modelo para que su salida sea, en esos casos, la misma que para las entradas modificadas.

La propagación tangencial y la ampliación del conjunto de entrenamiento utilizan transformaciones diseñadas específicamente, haciendo que el modelo aprendido sea invariante a ciertas direcciones de cambio en las entradas. El *backpropagation* doble y el entrenamiento con adversario hacen que el modelo sea invariante en todas las direcciones, siempre que los cambios correspondan sólo a pequeñas perturbaciones en las entradas.

Así pues, la ampliación del conjunto de entrenamiento usando transformaciones definidas manualmente puede interpretarse como la versión no infinitesimal de la propagación tangencial, mientras que el entrenamiento con adversario puede verse como la versión no infinitesimal del *backpropagation* doble. Una vez más, no todo es tan novedoso como podría parecer a primera vista.

Pese a sus antecedentes formales, que nunca salieron realmente del mundo académico, el entrenamiento con adversario resulta fascinante, tanto por sus implicaciones en temas de seguridad como por las posibilidades que sugiere a la hora de encontrar nuevas formas de hacer robusto el entrenamiento de redes neuronales. Para el que diseña redes neuronales, el entrenamiento con adversario ofrece nuevas vías de regularización que no requieren la realización de modificaciones, más o menos arbitrarias, sobre las funciones de coste con las que se entrena la red. Para el que se especializa en temas de seguridad, le abre la puerta al estudio de vulnerabilidades en las técnicas de aprendizaje automático y en todos los sistemas reales que las utilizan internamente.

Estudiantes y profesores

Antes de cerrar este apartado dedicado al conjunto de entrenamiento utilizado para ajustar los parámetros de una red neuronal artificial, comentemos otras estrategias que se pueden emplear para guiar el entrenamiento de redes neuronales en la dirección que deseemos, igual que un buen profesor debería ayudar a sus estudiantes a aprender sobre un tema, facilitándoles la tarea en lugar de poniéndoles obstáculos.

La destilación de conocimiento [*KD, Knowledge Distillation*] se propuso como una forma de comprimir modelos, conseguir un modelo más

²⁸² Harris Drucker y Yann LeCun. Double backpropagation increasing generalization performance. En *IJCNN'1991 International Joint Conference on Neural Networks, Seattle*, volume 2, pages 145–150, Jul 1991. DOI: 10.1109/IJCNN.1991.155328

²⁸³ Harris Drucker y Yann LeCun. Improving generalization performance using double backpropagation. *IEEE Transactions on Neural Networks*, 3(6): 991–997, Nov 1992. ISSN 1045-9227. DOI: 10.1109/72.165600

compacto a partir de un ensemble completo. Entrenar un ensemble, habitualmente, es una forma sencilla de mejorar el rendimiento de cualquier algoritmo de aprendizaje automático: múltiples modelos, siempre que no se equivoquen de la misma forma, obtienen mejores resultados que uno solo. La destilación de conocimiento consiste en conseguir un único modelo a partir de un ensemble.²⁸⁴ El ensemble realiza las funciones de profesor del modelo que realmente nos interesa, que es el estudiante. ¿Por qué no entrenar directamente al estudiante a partir del conjunto de entrenamiento? Porque se pretende que el estudiante aprenda no sólo la información proporcionada por las etiquetas del conjunto de entrenamiento, sino los matices que hayan sido capaces de captar los modelos que forman parte del ensemble que ejerce de profesor. De esta forma, se consigue comprimir un ensemble completo, el profesor, en un único modelo, el estudiante que aprende del profesor. ¿Cómo se entrena el estudiante? Haciendo que prediga no sólo las salidas reflejadas en el conjunto de datos de entrenamiento, sino también las predicciones que realiza el profesor.

El uso de los múltiples modelos del ensemble obtiene mejores resultados, pero puede ser demasiado costoso computacionalmente. Si conseguimos que un único modelo se comporte como el ejemplo en su conjunto, ese modelo se puede utilizar en sustitución del ensemble, de forma mucho más eficiente y con resultados similares. El modelo de destilación del conocimiento consiste en:

- Un profesor T con salidas $y_T = \text{softmax}(z_T)$, donde softmax es una función que genera una distribución de probabilidad a partir de los niveles de activación pre-softmax z_T correspondientes a las distintas clases del problema. Si el profesor es una sola red, z_T corresponde a las entradas netas de las neuronas de la capa de salida de la red. Si es un ensemble, y_T o z_T se obtienen promediando las salidas de diferentes redes (usando la media aritmética para y_T , la geométrica para z_T).
- Un estudiante S , que es una red con parámetros w_S y probabilidades de salida $y_S = \text{softmax}(z_S)$, donde z_S es la salida pre-softmax del estudiante. La red del estudiante se entrena de forma que su salida y_S sea similar a la salida del profesor y_T , además de las etiquetas reales del conjunto de entrenamiento y . Dado que la salida del profesor y_T ya se parecerá bastante a la salida deseada y , se añade un parámetro de relajación $\tau > 1$ para suavizar la salida del profesor, de forma que proporcione más información acerca de la similitud de un ejemplo de entrada a otras clases además de la clase con mayor probabilidad (para la que la salida y_T ya será prácticamente igual a 1). La relajación de la salida del profesor, y_T^τ , se aplica también a la salida del estudiante, y_S^τ ,

²⁸⁴ Geoffrey Hinton, Oriol Vinyals, y Jeff Dean. Distilling the Knowledge in a Neural Network. *NIPS 2014 Deep Learning and Representation Learning Workshop*, arXiv:1503.02531, 2014a. URL <https://arxiv.org/abs/1503.02531>

para poder compararla con la del profesor durante su entrenamiento:

$$\begin{aligned}y_T^\tau &= \text{softmax}(z_T / \tau) \\y_S^\tau &= \text{softmax}(z_S / \tau)\end{aligned}$$

La red del estudiante se entrena para minimizar la siguiente función de pérdida (o coste):

$$L_{KD}(w_S) = H(y, y_S) + \lambda H(y_T^\tau, y_S^\tau)$$

donde H es la entropía cruzada y λ es un parámetro ajustable para equilibrar ambas entropías cruzadas, la que corresponde al entrenamiento del estudiante con el conjunto de datos de entrenamiento y la que hace que el estudiante aprenda también la salida suavizada del profesor.

El modelo de destilación del conocimiento se propuso inicialmente para construir redes (estudiantes) a partir de ensambles (profesores) de redes con una arquitectura similar, obteniendo una red única a partir de un ensemble. Esta red es mucho más económica en cuanto a los recursos computacionales que necesita, lo que puede resultar útil en múltiples aplicaciones.

La misma estrategia se puede utilizar para entrenar redes individuales cuyas características las hacen difíciles de entrenar. En primer lugar, se entrena una red poco profunda pero ancha, más fácil de entrenar, a partir del conjunto de datos disponible. A continuación, esa red se utiliza como profesor de una segunda red más profunda que sería difícil de entrenar directamente a partir del conjunto de entrenamiento: el estudiante. La red del estudiante puede ser mucho más profunda y más delgada (con menos neuronas por capa) que la red del profesor. Al ser más esbelta, recibe el nombre de FitNet.²⁸⁵

El entrenamiento de una FitNet, delgada y profunda, sería mucho más difícil sin la colaboración del profesor. Sin embargo, una red profunda bien entrenada puede generar mejores resultados que una red poco profunda y más ancha. Además, si la hacemos lo suficientemente delgada, tendrá muchos menos parámetros que su profesor, por lo que su coste computacional también será menor cuando la usemos. La red que ejerce de profesor contribuye a hacer más sencillo el entrenamiento del estudiante. Le da pistas acerca de cuál debería ser su comportamiento, de ahí que esta estrategia también reciba el apelativo de entrenamiento basado en pistas [*hint-based training*]. Las pistas las podemos utilizar tanto en la capa de salida de la red, como en algunas de sus capas intermedias de la red, en las que haremos que el comportamiento del estudiante sea similar al comportamiento de las capas correspondientes del profesor. Las pistas proporcionadas por el profesor son las salidas de las neuronas de las capas que decidimos utilizar, lo que nos puede ayudar a entrenar redes estudiante mucho más profundas que las redes profesor. Redes que, de

La entropía cruzada, en el caso discreto, se define simplemente como $H(p, q) = -\sum_x p(x) \log q(x)$. Es una forma de medir las diferencias existentes entre dos distribuciones de probabilidad: la longitud esperada, en bits, del mensaje necesario por caso cuando se codifica de acuerdo a una distribución incorrecta Q pero los datos realmente siguen una distribución auténtica P .

²⁸⁵ Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, y Yoshua Bengio. FitNets: Hints for Thin Deep Nets. *ICLR'2015*, arXiv:1412.6550, 2015. URL <http://arxiv.org/abs/1412.6550>

otra forma, serían muy difíciles de entrenar utilizando sólo el gradiente descendente estocástico a partir de la señal de error en la capa de salida de la red.

El aprendizaje basado en pistas, propuesto originalmente por Yaser Abu-Mostafa, de Caltech, consiste básicamente en incluir información adicional [*prior information*] en el aprendizaje de una función $f(x)$. Usualmente, en el aprendizaje supervisado consiste exclusivamente en aprender a partir de la información proporcionada por los ejemplos del conjunto de entrenamiento: la salida y deseada dada una entrada x . Las pistas incorporan información adicional en el proceso de aprendizaje de $f(x)$.^{286,287} Esa información puede incluir propiedades de invarianza, como en la ampliación del conjunto de entrenamiento, o cualquier otra señal que nos ayude a acelerar la búsqueda de una buena aproximación de $f(x)$ y mejorar la calidad de dicha estimación, que es lo que se hace con la ayuda de un profesor en las FitNets.

El entrenamiento basado en pistas con destilación del conocimiento puede verse como una forma particular de aprendizaje curricular [*curriculum learning*].²⁸⁸ El aprendizaje curricular consiste en seleccionar adecuadamente la secuencia de conjuntos de entrenamiento con los que se entrena un modelo, comenzando por los casos más sencillos y refinando progresivamente su aprendizaje hasta llegar a los casos más complejos. El objetivo es que el modelo aprenda de forma similar a como un estudiante progresaría a lo largo de un plan de estudios bien diseñado. El aprendizaje curricular puede acelerar la convergencia del algoritmo de aprendizaje y, potencialmente, mejorar la capacidad de generalización del modelo entrenado.

De hecho, algunas extensiones del aprendizaje curricular ya habían sugerido el uso de pistas que guiasen el entrenamiento de las capas ocultas de una red multicapa, de forma que se facilitase el aprendizaje de redes profundas.²⁸⁹ El aprendizaje basado en pistas ofrece una alternativa viable en situaciones en las que fallan los algoritmos estándar de entrenamiento de redes neuronales. Su principal limitación es que la estrategia concreta suele diseñarse de forma *ad hoc*, en función del problema concreto al que nos enfrentemos.

Inicialización de la red

Analicemos, a continuación, uno de los factores a los que normalmente no se presta demasiada atención a la hora de entrenar una red neuronal: su inicialización. Pese a lo que pueda parecer, una inicialización correcta de la red puede suponer la diferencia entre el éxito o el fracaso del algoritmo utilizado para su entrenamiento.

²⁸⁶ Yaser S. Abu-Mostafa. Learning from hints. *Journal of Complexity*, 10(1):165 – 178, 1994. ISSN 0885-064X. DOI: jcom.1994.1007

²⁸⁷ Yaser S. Abu-Mostafa. Hints. *Neural Computation*, 7(4):639–671, 1995. DOI: 10.1162/neco.1995.7.4.639

²⁸⁸ Yoshua Bengio, Jérôme Louradour, Ronan Collobert, y Jason Weston. Curriculum learning. En Andrea Pohoreckyj Danyluk, Léon Bottou, y Michael L. Littman, editores, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 41–48. ACM, 2009. ISBN 978-1-60558-516-1. DOI: 10.1145/1553374.1553380

²⁸⁹ Çağlar Gülcehre y Yoshua Bengio. Knowledge Matters: Importance of Prior Information for Optimization. *ICLR'2013*, arXiv:1301.4083, 2013. URL <http://arxiv.org/abs/1301.4083>

Inicialización de los pesos de la red

Si cogemos cualquier red neuronal e inicializamos los pesos de todas las neuronas de la red exactamente de la misma forma, la red neuronal será incapaz de aprender absolutamente nada, algo que se puede comprobar fácilmente. La corrección realizada sobre los pesos, derivada del gradiente de la señal de error, será siempre la misma para todas las neuronas de una misma capa. Todos sus pesos se ajustarán en el mismo sentido, en una misma cantidad, y seguirán siendo iguales indefinidamente.

Cuando, dentro de una misma capa, dos neuronas tienen exactamente los mismos pesos, reciben la misma señal de error de capas posteriores de la red, la misma entrada de las capas anteriores y se les atribuye la misma aportación al error observado. En otras palabras, serán incapaces de aprender características diferentes. Podríamos sustituir todas las neuronas de la capa por una única neurona y obtendríamos el mismo resultado.

Así pues, es imprescindible que inicialicemos todos los pesos de una red neuronal utilizando valores aleatorios para romper la simetría de la red.

Usualmente, se utilizan como base los valores provenientes de un generador de números pseudoaleatorios. A partir de ahí, se generan valores aleatorios de acuerdo a una distribución uniforme o normal.^{290,291} No parece importar demasiado el tipo de distribución utilizada para generar los valores iniciales de los pesos, si bien la escala de los pesos utilizados sí que influye decisivamente en el entrenamiento de la red. Se recomienda que los pesos se inicialicen con valores pequeños, especialmente si utilizamos funciones sigmoidales de activación, con el objetivo de evitar su saturación, que ralentizaría el entrenamiento de la red.

Al ajustar los pesos de una red neuronal, un algoritmo de optimización como el gradiente descendente estocástico realiza pequeños cambios incrementales sobre sus valores. Esto hace recomendable que inicialicemos los pesos con valores cercanos a cero, tanto positivos como negativos, para facilitar que, dada una señal de error, unos se actualicen en un sentido y otros en el sentido opuesto.

No obstante, si siempre usamos pesos muy bajos puede que todos los pesos asociados a una neurona acaben siendo prácticamente nulos, con lo que la neurona resultaría redundante en la red, al no realizar aportación alguna. Esto nos puede hacer pensar que deberíamos inicializar los pesos de la red con valores más altos, que tengan un mayor efecto a la hora de romper la simetría de la red, eviten la aparición de neuronas redundantes y aseguren la propagación de la señal de error en *backpropagation*, mitigando aunque sea parcialmente fenómenos como la desaparición del gradiente. Sin embargo, si empleamos valores demasiado altos en la inicialización de los pesos de la red, esto puede hacer que se saturen las neuronas

²⁹⁰ George E. P. Box y Mervin E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 06 1958. DOI: 10.1214/aoms/1177706645

²⁹¹ William H. Press, Saul A. Teukolsky, William T. Vetterling, y Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007. ISBN 0521880688

en la propagación hacia adelante de la señal de entrada de la red, lo que ocasiona la desaparición del gradiente en la propagación hacia atrás del error e impide el correcto entrenamiento de la red. En algunas arquitecturas de red, como las redes recurrentes que incluyen ciclos, unos pesos elevados pueden ocasionar la inestabilidad de la red, la cual puede desembocar en un comportamiento caótico. Bajo un régimen caótico, un sistema dinámico es extremadamente sensible a sus condiciones iniciales y a pequeñas perturbaciones que puedan producirse en sus entradas, como recordará de la mariposa que al agitar sus alas acaba ocasionando la formación de un huracán.²⁹²

¿Cuál sería una buena elección para los valores iniciales de los pesos de la red? Si son demasiado elevados, pueden aparecer los problemas asociados a la saturación prematura de las neuronas de la red, o incluso problemas más graves como el comportamiento caótico del algoritmo de aprendizaje. Si son demasiado bajos, estaremos moviéndonos en una zona de la superficie de error demasiado plana, cercana al origen, en la que la señal que se propaga a través de la red se va reduciendo en cada capa hasta resultar demasiado pequeña para ser útil. Debemos buscar un punto intermedio entre esos dos extremos.

Imaginemos que tenemos una red multicapa que utiliza la tangente hiperbólica como función de activación. Asumamos que el sesgo de las neuronas es cero, por lo que su entrada neta es $z = w \cdot x$, con $w_0 = 0$. Como recomendamos al describir el preprocesamiento de los datos de entrada, las entradas aplicadas a la red estarán tipificadas, con media cero y varianza unitaria. Asumamos, además, que las entradas están decorreladas, por lo que $E[x_i x_j] = 0$ si $i \neq j$ y, obviamente, $E[x_i x_i] = 1$. Partamos de unos pesos inicializados aleatoriamente utilizando una distribución uniforme con media cero, $\mu_w = E(w_{ij}) = 0$, y varianza $\sigma_w^2 = E[(w_{ij} - \mu_w)^2] = E[w_{ji}^2]$. En una situación como la descrita, podemos obtener la media y varianza de la entrada neta de una neurona:

$$\mu_z = E[\sum w_{ij} x_i] = \sum E[w_{ij}] E[x_i] = 0$$

$$\sigma_z^2 = E[(z - \mu_z)^2] = E[z^2] = n\sigma_w^2$$

donde n es el número de entradas de la neurona.

A partir de estos resultados, podemos diseñar una estrategia de inicialización de los pesos de tal forma que la entrada neta de la neurona se mantenga en la zona lineal de su función de activación sigmoidal, sin llegar a la zona en la que la neurona se satura. Para una neurona que utilice la tangente hiperbólica con los parámetros recomendados en una sección anterior ($a = 1.7159$ y $b = 2/3$), podemos conseguir nuestro objetivo si establecemos $\sigma_z = 1$. De acuerdo con las ecuaciones que acabamos de obtener, los pesos deben inicializarse con media 0 y desviación $\sigma_w = 1/\sqrt{n}$.

²⁹² James Gleick. *Chaos: Making a New Science* . Viking Books, 1987. ISBN 0670811785

En conclusión, si inicializamos los pesos con una distribución uniforme, los pesos sinápticos deben seleccionarse de forma que tengan media cero y varianza igual al recíproco de su número de entradas ($1/n$).

¿Por qué hacemos que el rango de los pesos dependa de su número de entradas, también conocido como *fan-in* tomando la terminología habitual de los circuitos electrónicos? Si una neurona tiene muchas conexiones de entrada, un *fan-in* elevado, pequeños cambios en todos sus pesos pueden hacer que nos pasemos en el ajuste de los pesos cuando aplicamos una corrección basada en el gradiente del error: el efecto de un cambio Δw en cada uno de los pesos se traduce en un cambio de magnitud $n\Delta w$ en total cuando consideramos la entrada neta de la neurona.

Normalmente, querremos utilizar pesos más pequeños cuanto mayor sea el *fan-in* de una neurona, por lo que inicializaremos los pesos aleatoriamente de forma proporcional a $1/\sqrt{n}$. La varianza de una distribución uniforme $U(a, b)$ es $(b - a)^2/12$ y su media es $(a + b)/2$. Si deseamos que la media de los pesos sea cero, entonces $a = -b$. Dada una distribución uniforme $U(-a, a)$, querremos que su varianza, $a^2/3$, sea igual a $1/n$. Por tanto, los pesos los debemos inicializar de acuerdo a una distribución $U(-\sqrt{3/n}, \sqrt{3/n})$, en la que los pesos son proporcionales al recíproco de la raíz cuadrada del *fan-in* de la neurona ($1/\sqrt{n}$).

Si, en vez de utilizar una distribución uniforme, optamos por una distribución gaussiana para inicializar los pesos de la red, sólo tenemos que generar números aleatorios de acuerdo a una distribución normal de media 0 y varianza $1/n$: $N(0, 1/n)$. El efecto de hacer que la desviación estándar de la gaussiana sea $1/\sqrt{n}$ es estrecharla conforme aumenta el número de entradas de la neurona (y hacer que su pico sea más pronunciado en torno a su media). La entrada neta de la neurona, z , tendrá una distribución normal, de varianza 1, lo que hará poco probable que la neurona se sature y facilitará el entrenamiento de la red.

Hemos deducido por qué se suele recomendar que, asumiendo que los datos de entrenamiento se normalizan y se usa la función de activación $f(z) = 1.7159 \tanh(2z/3)$, los pesos se deben inicializar aleatoriamente utilizando una distribución (p.ej. uniforme) con media $\mu_w = 0$ y desviación estándar $\sigma_w = 1/\sqrt{n}$, donde n es el *fan-in* de la neurona.

No es la única estrategia que podríamos haber utilizado para inicializar los pesos de la red:

- *Método de Nguyen y Widrow*²⁹³

Es un método de inicialización de los pesos que corresponden a las conexiones sinápticas que conectan la capa de entrada con las unidades ocultas de una red con una única capa oculta. Está diseñado para mejorar la capacidad de aprendizaje de las neuronas ocultas, haciendo que la entrada neta de las neuronas ocultas estén en el rango en el que aprenden con mayor facilidad (asumiendo, de nuevo, que se utiliza la

Asumiendo entradas en el intervalo [0, 1] y aplicando un razonamiento análogo, llegaríamos a un resultado similar: $U(-3/\sqrt{n}, 3/\sqrt{n})$.

Lodewyk F. A. Wessels y Etienne Barnard. Avoiding false local minima by proper initialization of connections. *IEEE Transactions on Neural Networks*, 3(6):899–905, Nov 1992. ISSN 1045-9227. doi: 10.1109/72.165592

²⁹³ Derrick Nguyen y Bernard Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. En *IJCNN'1990 International Joint Conference on Neural Networks, Washington DC*, pages III:21–26, June 1990. doi: 10.1109/IJCNN.1990.137819

tangente hiperbólica como función de activación).

En primer lugar, define un factor de escala $\beta = 0.7 \sqrt[3]{p}$, donde p es el número de neuronas ocultas y n el número de neuronas de la capa de entrada.

A continuación, inicializa la matriz de pesos w_{ij} con valores aleatorios entre -0.5 y +0.5 (o entre $-\gamma$ y $+\gamma$).

Acto seguido calcula la norma o magnitud del vector asociado a cada neurona oculta $\|w_j\| = \sqrt{\sum_i w_{ij}^2}$. Esa magnitud y el factor de escala β se emplean para normalizar el vector de pesos asociados a cada neurona $w_{ij} = \beta w_{ij} / \|w_j\|$.

Para el sesgo de las neuronas, b_j o w_{0j} , se escoge un valor aleatorio entre $-\beta$ y $+\beta$.

El método de Nguyen y Widrow es el utilizado, por ejemplo, en la función `initnw` incluida en el *toolbox* de Matlab para redes neuronales.

Frente a una inicialización puramente aleatoria de los pesos, evita la aparición de neuronas redundantes que desperdician parte de la capacidad de la red neuronal y permite que el entrenamiento de la red se realice con mayor velocidad.

■ Inicialización de Xavier²⁹⁴

Frente a la recomendación tradicional de inicializar los pesos tomando muestras de una distribución uniforme $U(-1/\sqrt{n}, 1/\sqrt{n})$, Xavier Glorot y Yoshua Bengio sugieren modificar los límites la distribución uniforme: $U(-\sqrt{6/(n_{in} + n_{out})}, \sqrt{6/(n_{in} + n_{out})})$, donde n_{in} es el número de entradas de una capa y n_{out} el número de neuronas que reciben las salidas de la capa (el número de neuronas en la siguiente capa de la red). Es decir, además del *fan-in* de las neuronas, el método de Xavier tiene en cuenta su *fan-out*.

El método de Xavier está diseñado para llegar a un compromiso entre inicializar todas las capas de la red de forma que tengan la misma varianza e inicializar las distintas capas de la red de forma que el gradiente del error también tenga la misma varianza. La fórmula se obtiene asumiendo que la red consiste en una simple multiplicación de matrices, ignorando sus componentes no lineales. Una suposición poco realista, pero que da lugar a una heurística que funciona razonablemente bien en la práctica. De hecho, fue uno de los factores que permitió el entrenamiento de redes neuronales con múltiples capas ocultas usando gradiente descendente y *backpropagation* sin necesidad de recurrir a un algoritmo *greedy* que vaya entrenando los parámetros de la red capa a capa.

En la propagación hacia adelante de la señal de la red, ya hemos visto que conviene que los pesos se inicialicen con una distribución de media 0 y varianza $1/n_{in}$. Utilizando un argumento similar para la señal de error que se propaga hacia atrás con *backpropagation*, se llega a que la varianza de los pesos debe ser $1/n_{out}$ para que la varianza

²⁹⁴ Xavier Glorot y Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. En Yee Whye Teh y D. Mike Titterington, editores, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256, 2010. URL <http://jmlr.org/proceedings/papers/v9/glorot10a>

del gradiente de entrada y de salida sea la misma (lo que facilita que las distintas capas de la red puedan aprender al mismo ritmo).

Las dos restricciones anteriores, que la varianza sea $1/n_{in}$ en la propagación hacia adelante y $1/n_{out}$ en la propagación hacia atrás sólo puede verificarse si $n_{in} = n_{out}$, por lo que Glorot y Bengio llegan a un compromiso usando la media de ambos valores, por lo que la varianza en la inicialización de los pesos debe ser $\sigma_w^2 = 2/(n_{in} + n_{out})$. De esa varianza se derivan los límites de la distribución uniforme con la que se inicializan los pesos de la red: $U(-\sqrt{6/(n_{in} + n_{out})}, \sqrt{6/(n_{in} + n_{out})})$ para la tangente hiperbólica y $U(-4\sqrt{6/(n_{in} + n_{out})}, 4\sqrt{6/(n_{in} + n_{out})})$ para la función logística.

Este método de inicialización aparece referenciado en algunas de las bibliotecas utilizadas en *deep learning*, como es el caso de Caffe, donde uno se puede encontrar líneas como la siguiente en la definición de las capas de una red: `weight_filler { type: "xavier"}`. Sin embargo, Caffe sólo utiliza n , tal vez para facilitar su implementación en la práctica, por lo que, estrictamente, su método de inicialización no debería denominarse `xavier`.

■ Inicialización de unidades ReLU²⁹⁵

El método de Xavier asumía que la red estaba compuesta de neuronas lineales, ya que se tenía como objetivo que la inicialización de los pesos hiciera que la activación de las neuronas se mantuviese en su zona lineal (donde, además, el gradiente de la función de activación es cercano a 1). Esas suposiciones no son válidas si utilizamos unidades rectificadas lineales, ReLU.

Cuando utilicemos unidades ReLU en nuestra red neuronal, investigadores de Microsoft Research sugieren que dupliquemos la varianza en la inicialización de los pesos: $\sigma_w^2 = \frac{2}{n}$ donde n , una vez más, es el *fan-in* de las neuronas de la red.

Este ajuste, que se traduce en una ampliación de los intervalos usados en las distribuciones de probabilidad con las que se establecen los valores iniciales de los pesos, tiene sentido desde el punto de vista intuitivo. Una unidad lineal rectificada se mantiene inactiva para la mitad de sus entradas, por lo que hay que doblar la varianza de su entrada para mantener constante la varianza de su señal de salida.

Por tanto, en la práctica, los pesos de las unidades ReLU los inicializaremos con una distribución uniforme $U(-\sqrt{6/n}, \sqrt{6/n})$ o con una distribución normal $N(0, 2/n)$. En código, la expresión necesaria será de la forma `w = 2 * (random.uniform() - 0.5) * sqrt(6.0/n)` si empleamos una distribución uniforme, `w = random.normal() * sqrt(2.0/n)` si usamos una distribución normal.

Existen otras muchas estrategias de inicialización de los pesos de una red neuronal. Por ejemplo, podría interesarnos realizar una inicialización

²⁹⁵ Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv e-prints*, 2015a. URL <http://arxiv.org/abs/1502.01852>

dispersa [*sparse initialization*]²⁹⁶ en la que cada neurona se inicializa para tener exactamente k pesos distintos de cero. De esta forma se mantiene la entrada neta total de la neurona independiente de su *fan-in*, sin tener que hacer que la magnitud de los pesos de cada neurona tenga que disminuir conforme aumenta el *fan-in*. La inicialización dispersa crea una mayor diversidad inicial entre las neuronas de una capa. Sin embargo, puede que necesitemos muchas iteraciones del gradiente descendente para corregir valores iniciales “incorrectos”

Otros métodos de inicialización son más sofisticados desde el punto de vista matemático y están diseñados para facilitar el entrenamiento de redes neuronales profundas, con muchas capas ocultas. Andrew Saxe, de la Universidad de Stanford, emplea matrices ortogonales aleatorias para mantener la red en un régimen dinámico en el que los niveles de activación aumentan hacia adelante y los gradientes aumentan hacia atrás.²⁹⁷ David Sussillo utiliza caminos aleatorios [*random walks*] para preservar la magnitud de los gradientes, evitando de esa forma fenómenos como la desaparición o la explosión del gradiente, lo que le permite entrenar redes de hasta 1000 capas, nada menos.²⁹⁸

Técnicas como la inicialización LSUV [*Layer-Sequential Unit-Variance*]²⁹⁹ se han utilizado para entrenar redes muy profundas sin necesidad de recurrir a estrategias más complejas de entrenamiento como las utilizadas en FitNets³⁰⁰ o redes Highway.³⁰¹ Su éxito parece sugerir que una buena inicialización de los pesos de la red es fundamental para solucionar muchos de los problemas tradicionalmente asociados al entrenamiento de redes profundas. Hasta el punto de llegar a afirmar que todo lo que se necesita es... una buena inicialización.

Aunque algunas heurísticas han demostrado funcionar excepcionalmente bien, hay expertos que sugieren que, si nuestros recursos computacionales nos lo permiten, tratemos la escala inicial de los pesos para cada capa de la red como un hiperparámetro más. Sin embargo, esta opción no resulta demasiado atractiva, por su coste computacional, necesario para evaluar cada configuración de la red utilizando un conjunto de validación independiente (el conjunto de datos, separado de los conjuntos de entrenamiento y de prueba, que se emplea para ajustar los valores de los hiperparámetros).

Resulta mucho más sencillo construir un modelo inicial y, sobre ese modelo, analizar el funcionamiento de la red sobre un lote de los datos de los que dispongamos. Sobre ese lote podemos monitorizar los valores de activación de las neuronas de las distintas capas de la red. Si los pesos son demasiado pequeños, las activaciones de las neuronas se desvanecerán conforme se propaga la señal de entrada a través de la red. Identificando la primera capa en la que las activaciones son demasiado reducidas e incrementando la escala de sus pesos, conseguimos una estrategia sencilla de implementar que consigue el objetivo deseado: mantener unos niveles de

²⁹⁶ James Martens. Deep Learning via Hessian-free Optimization. En *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML’10, pages 735–742. Omnipress, 2010. ISBN 978-1-60558-907-7. URL <http://icml2010.haifa.il.ibm.com/papers/458.pdf>

²⁹⁷ Andrew M. Saxe, James L. McClelland, y Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *ICLR’2014*, arXiv:1312.6120, 2014. URL <http://arxiv.org/abs/1312.6120>

²⁹⁸ David Sussillo. Random walk initialization for training very deep feed-forward networks. *arXiv e-prints*, arXiv:1412.6558, 2014. URL <http://arxiv.org/abs/1412.6558>

²⁹⁹ Dmytro Mishkin y Jiri Matas. All you need is a good init. *ICLR’2016*, arXiv:1511.06422, 2016. URL <http://arxiv.org/abs/1511.06422>

³⁰⁰ Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, y Yoshua Bengio. FitNets: Hints for Thin Deep Nets. *ICLR’2015*, arXiv:1412.6550, 2015. URL <http://arxiv.org/abs/1412.6550>

³⁰¹ Rupesh Kumar Srivastava, Klaus Greff, y Jürgen Schmidhuber. Training very deep networks. En Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, y Roman Garnett, editores, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2377–2385, 2015c. URL <http://papers.nips.cc/paper/5850-training-very-deep-networks>

activación razonables en todas las capas de la red. Si el aprendizaje sigue siendo demasiado lento, Goodfellow, Bengio y Courville recomiendan comprobar los rangos y las derivadas de los gradientes propagados hacia atrás, además de las activaciones propagadas hacia adelante, con el objetivo de ir ajustando los pesos de las distintas capas de la red hasta conseguir el comportamiento deseado.

Si preferimos optar por un método que no requiera nuestra intervención manual, el abanico de técnicas heurísticas que se han propuesto casi no tiene límites. Podemos inicializar los pesos aleatoriamente en el intervalo $[-0.77, 0.77]$ a partir de los resultados empíricos obtenidos en cientos de miles de simulaciones, sin entrar en disquisiciones de tipo teórico.³⁰² Podemos inicializar los pesos en un intervalo $[-c, c]$, con $c > 0$, y luego reescalar los pesos individualmente para asegurarnos de que la neurona se mantiene activa de acuerdo al rango dinámico de su función de activación.³⁰³ Podemos recurrir a métodos estadísticos con nombres más o menos afortunados, como SCAWI [*Statistically-Controlled Activation Weight Initialization*], que realiza un análisis estadístico para obtener la amplitud máxima de los pesos iniciales.³⁰⁴ Incluso podemos recurrir a argumentos de tipo geométrico.³⁰⁵

Cualquiera de las técnicas heurísticas mencionadas puede servirnos para ajustar adecuadamente la escala inicial de los pesos de nuestra red y facilitar su entrenamiento usando técnicas convencionales de optimización basadas en el gradiente descendente y *backpropagation*.

Inicialización de los sesgos de las neuronas

Cualquiera de los criterios heurísticos de la sección anterior podríamos aplicarlos a la inicialización de los sesgos [*biases*] de las neuronas. En realidad, el uso de sesgos puede ser opcional en muchos problemas, por lo que nuestra red tal vez carezca de ellos.

La estrategia más habitual consiste en inicializar los sesgos de todas las neuronas de la red con una constante elegida de forma heurística e inicializar aleatoriamente sólo los pesos de la red. La inicialización aleatoria de los pesos ya rompe la simetría de la red, lo que resulta suficiente para que dejemos que sea el gradiente descendente el responsable de establecer un valor adecuado para los sesgos de las distintas neuronas.

Es muy común inicializar los sesgos a 0, una inicialización que suele ser compatible con casi todas las heurísticas de inicialización de los pesos. No obstante, existen algunas situaciones excepcionales en las que los sesgos se inicializan con valores distintos de cero:

- *En unidades de salida:* En problemas de clasificación, se pueden inicializar los sesgos de las neuronas para que la salida de la red refleje las probabilidades asociadas a las diferentes clases del problema (asumiendo que los pesos iniciales son lo suficientemente pequeños como

³⁰² G. Thimm y E. Fiesler. High-order and multilayer perceptron initialization. *IEEE Transactions on Neural Networks*, 8(2):349–359, Mar 1997. ISSN 1045-9227. DOI: 10.1109/72.557673

³⁰³ Sean McLoone, Michael D. Brown, George W. Irwin, y Gordon Lightbody. A hybrid linear/nonlinear training algorithm for feedforward neural networks. *IEEE Transactions on Neural Networks*, 9(4):669–684, Jul 1998. ISSN 1045-9227. DOI: 10.1109/72.701180

³⁰⁴ G. P. Drago y S. Ridella. Statistically controlled activation weight initialization (SCAWI). *IEEE Transactions on Neural Networks*, 3(4):627–631, Jul 1992. ISSN 1045-9227. DOI: 10.1109/72.143378

³⁰⁵ J. Y. F. Yam y T. W. S. Chow. Feed-forward networks training speed enhancement by optimal initialization of the synaptic coefficients. *IEEE Transactions on Neural Networks*, 12(2):430–434, Mar 2001. ISSN 1045-9227. DOI: 10.1109/72.914538

para que la salida de la red venga determinada sólo por el sesgo de las unidades de salida).

- *Unidades ReLU:* Para evitar que las unidades lineales rectificadas se saturen demasiado al inicializar los pesos de la red, se les suele asignar un pequeño sesgo positivo (p.ej. 0.1 en vez de 0.0). De esta forma, se consigue que las unidades ReLU se mantengan activas inicialmente para la mayor parte de los datos del conjunto de entrenamiento, lo que permite la propagación del gradiente del error hacia atrás. En la práctica, no está demasiado claro si esta estrategia de inicialización ayuda realmente o no.
- *Unidades de control:* En ciertos tipos de redes, hay unidades que controlan si otra unidad participa o no activamente en el procesamiento de los datos de entrada. Si la unidad de control c genera una salida que se combina con la de otra unidad h para obtener una señal ch , la unidad c hace de “puerta”: determina si la salida final ch es h o 0. En esos casos, se suele establecer el sesgo de la unidad de control para que su salida sea 1 tras la inicialización de la red. Por ejemplo, se establece el sesgo a 1 en las “puertas del olvido” de las redes recurrentes de tipo LSTM.³⁰⁶

Otra posibilidad igualmente válida en muchas ocasiones consiste en inicializar aleatoriamente los sesgos de la red utilizando valores aleatorios de acuerdo a una distribución normal de media 0 y desviación estándar 1. Esta inicialización es poco probable que haga que las neuronas sigmoidales se saturen, por lo que no suele dar problemas. En general, no importa cómo inicialicemos los sesgos, siempre que prevengamos los problemas derivados de la saturación prematura de las neuronas de la red.

La estrategia recomendada para inicializar los sesgos es, por tanto, asumir que los pesos iniciales serán prácticamente nulos, por lo que estableceremos un valor adecuado para el sesgo en función de la salida inicial que deseemos conseguir, ignorando por completo los posibles efectos de los pesos y las entradas que reciba la red. El valor concreto del sesgo que resulte más adecuado dependerá del contexto (del problema que pretendamos resolver y del tipo de red que estemos entrenando).

Pre-entrenamiento de la red

Hemos visto que existen métodos de entrenamiento (p.ej. FitNets) y estrategias de inicialización (Xavier o LSUV) diseñadas específicamente para entrenar redes neuronales con múltiples capas ocultas. De hecho, durante mucho tiempo, no se supo cómo entrenar correctamente redes neuronales multicapa con más de una capa oculta, salvo para casos particulares como las redes convolutivas utilizadas en el procesamiento de imágenes.

³⁰⁶Rafal Jozefowicz, Wojciech Zaremba, y Ilya Sutskever. An empirical exploration of recurrent network architectures. En *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, volume 37 of *ICML’15*, pages 2342–2350. JMLR.org, 2015. URL <http://proceedings.mlr.press/v37/jozefowicz15.pdf>

Una capacidad de cálculo limitada, la escasa disponibilidad de conjuntos de datos etiquetados suficientemente grandes, el uso de redes neuronales demasiado pequeñas e inicializadas de forma poco razonable influyeron en que decayese el interés en las redes neuronales artificiales en los años 90. Al menos, hasta 2006, año en el que la publicación de tres trabajos de Geoffrey Hinton,³⁰⁷ Yoshua Bengio³⁰⁸ y Yann LeCun³⁰⁹ marcó el renacer de las redes neuronales artificiales y el despegue de lo que hoy conocemos como *deep learning*.

Las primeras estrategias de entrenamiento de redes neuronales profundas, con más de una capa oculta, fueron algoritmos *greedy* que iban entrenando las redes por capas, como el algoritmo de la torre de Stephen Gallant, propuesto para entrenar redes multicapa utilizando el algoritmo de aprendizaje del perceptrón.³¹⁰ Aunque su uso no esté tan extendido ahora como durante los primeros años del *deep learning*, conviene analizar en qué consistían esas estrategias iniciales de entrenamiento de redes profundas. No sólo por razones históricas, sino porque aún se pueden aprovechar en determinadas aplicaciones.

La idea original tras las primeras propuestas consistía en ir apilando capas para construir redes profundas de distintos tipos. Hinton apilaba máquinas de Boltzmann restringidas para formar redes de creencia profundas [*DBNs: Deep Belief Nets*] y Bengio generalizaba la estrategia de entrenamiento *greedy* por capas [*greedy layer-wise training*] para redes profundas. El entrenamiento por capas de una red procede de la siguiente forma. En primer lugar, se entrena los parámetros de una capa de la red, directamente a partir de los datos de entrada. A continuación, añadimos una segunda capa y utilizamos los niveles de activación de la primera capa, previamente entrenada, para entrenar los parámetros de la segunda capa de la red. El algoritmo *greedy* se repite iterativamente, para ajustar los parámetros de la capa $i + 1$ a partir de los parámetros previamente ajustados de las i capas que la preceden, formando una red profunda por apilamiento [*stacking*].

Esta estrategia genérica se puede utilizar para inicializar los parámetros de una red profunda, ya sea empleando aprendizaje supervisado o aprendizaje no supervisado:

- Pre-entrenamiento no supervisado [*unsupervised pretraining*]: Si disponemos de acceso a un conjunto de datos no etiquetado mucho más grande que el conjunto de datos etiquetado con el que pretendemos entrenar nuestra red, podemos construir un modelo no supervisado que reciba las mismas entradas y usar ese modelo como punto de partida para el entrenamiento de la red con los datos etiquetados de los que dispongamos.
- Pre-entrenamiento supervisado [*supervised pretraining*]: Tal vez podamos aprovechar que el problema que pretendemos resolver no es

³⁰⁷ Geoffrey E. Hinton, Simon Osindero, y Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006. DOI: 10.1162/neco.2006.18.7.1527

³⁰⁸ Yoshua Bengio, Pascal Lamblin, Dan Popovici, y Hugo Larochelle. Greedy layer-wise training of deep networks. En *NIPS'2006 Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 153–160, 2006a. URL <https://goo.gl/KyBc9x>

³⁰⁹ Marc'Aurelio Ranzato, Christopher S. Poultney, Sumit Chopra, y Yann LeCun. Efficient learning of sparse representations with an energy-based model. En *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 1137–1144, 2006. URL <https://goo.gl/pKBzV0>

³¹⁰ Stephen I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, June 1990. ISSN 1045-9227. DOI: 10.1109/72.80230

el único problema en el que se utilizan datos similares a los de nuestro conjunto de entrenamiento. En ese caso, podemos aprovechar el aprendizaje supervisado realizado para resolver otro problema relacionado y utilizar los parámetros de ese otro modelo para inicializar los del nuestro. Es lo que se conoce como aprendizaje por transferencia [*transfer learning*], una forma primitiva de aprendizaje por analogía.

Tanto con un enfoque no supervisado [*unsupervised pretraining*] como con un enfoque supervisado [*transfer learning*], podemos aprovechar procesos previos de entrenamiento para inicializar los parámetros de nuestra red. A continuación, aplicamos un algoritmo de ajuste de esos parámetros para adaptarlos a nuestro problema particular de aprendizaje.

Esta estrategia de inicialización permite que el algoritmo de entrenamiento de nuestra red converja más rápidamente que si nos limitásemos a una inicialización aleatoria de los pesos de la red, como alternativa a las técnicas de inicialización que hemos visto para establecer la escala adecuada de los pesos iniciales de la red.

Además, nuestro modelo, una vez ajustado tras su pre-entrenamiento, puede generalizar mejor en la práctica. ¿Por qué? Porque el proceso de pre-entrenamiento, tanto supervisado como no supervisado, le ha dado la oportunidad de establecer sus parámetros iniciales aprovechando las características de los conjuntos de datos reales. Formalmente, al usar pre-entrenamiento, los parámetros iniciales de la red se ajustan de acuerdo a la distribución de probabilidad real de los datos, p_{data} , en lugar de a una distribución aleatoria, ya sea normal o uniforme.

El pre-entrenamiento de una red neuronal permite compartir características extraídas de los datos entre diferentes modelos de aprendizaje, tanto supervisados como no supervisados. Se pueden ver las redes multicapa como mecanismos de aprendizaje de representaciones que extraen conjuntos jerárquicos de características. La última capa de la red es, a menudo, un simple clasificador lineal (p.ej. una capa *softmax*), especializado para resolver un problema concreto. Sus capas ocultas representan características extraídas de los datos que luego se utilizan en la capa de salida de la red para resolver el problema. Esas características, sin embargo, se pueden reutilizar en otros muchos problemas relacionados. A menudo, que un problema sea fácil o difícil de resolver depende de cómo representemos los datos. El pre-entrenamiento de una red nos puede proporcionar una representación de los datos que nos facilite el problema en el que realmente estemos interesados.

En el caso supervisado, la propuesta original de Bengio, cada etapa del algoritmo *greedy* involucraba un subconjunto de las capas de la red final: cada capa se entrenaba como una red tradicional, de una sola capa oculta, usando como entrada la salida de la capa oculta previamente entrenada. Otra alternativa, análoga al algoritmo de la torre de Gallant,

es utilizar tanto las salidas de las redes previamente entrenadas como los datos de entrada en cada capa que se añade a la red.³¹¹ Propuestas más recientes optan por entrenar inicialmente una red con múltiples capas a la que luego se le añaden múltiples capas intermedias hasta conseguir redes profundas capaces de resolver problemas extremadamente complejos.³¹²

El pre-entrenamiento supervisado, de forma natural, consigue extraer jerarquías de características en cada capa de la red. Utilizando como criterio el mismo que el asociado al problema supervisado que deseamos resolver, las características extraídas progresivamente nos harán más sencillo discriminar entre las distintas clases presentes en nuestros datos, más cuanto más ascendamos en la red. Idealmente, tal vez consigan que las clases acaben siendo linealmente separables. Las características aprendidas en el pre-entrenamiento podemos utilizarlas entonces como entrada de una capa final de la red, para construir un clasificador basado en una red neuronal multicapa, o como entrada de cualquier otro tipo de modelo de clasificación, p.ej. un clasificador k-NN. De hecho, podríamos utilizar otro tipo de modelo de clasificación para guiar el proceso completo de pre-entrenamiento de las distintas capas de la red, no tiene por qué ser un clasificador basado en una red neuronal. El tipo de características que se extraerán en el pre-entrenamiento tendrán diferentes propiedades en función del tipo de clasificador que utilicemos como capa final.³¹³

Cuando queremos resolver un problema complejo, ya sea porque la topología de la red la haga difícil de entrenar o porque el problema en sí sea muy difícil, podemos entrenar una red más simple que resuelva el problema y luego hacerla más compleja, como en el pre-entrenamiento supervisado, o bien entrenar la red para resolver un problema más sencillo y luego abordar el problema complejo, como se hace en el pre-entrenamiento no supervisado.

En el caso no supervisado, el pre-entrenamiento consiste en ajustar los parámetros de nuestro modelo aprovechándonos de la disponibilidad de enormes conjuntos de datos no etiquetados. Es relativamente habitual que dispongamos de relativamente pocos datos correctamente etiquetados pero que podamos acceder a conjuntos de datos mucho más grandes sin etiquetar. En tales situaciones, podemos intentar extraer buenos conjuntos de características de los datos no etiquetados y luego utilizarlos para resolver nuestro problema de aprendizaje supervisado, aun disponiendo de pocos ejemplos etiquetados. Al fin y al cabo, los animales, seres humanos incluidos, parece que aprendemos así, utilizando muy pocos ejemplos etiquetados de forma explícita y aprovechando todo el bagaje que acumulamos con nuestras experiencias previas. El cerebro parece, por tanto, capaz de aprovechar sin problemas su pre-entrenamiento.

Como en el pre-entrenamiento supervisado, el pre-entrenamiento no supervisado se puede realizar utilizando una estrategia *greedy* por capas. Cada iteración se utiliza para ajustar un subconjunto de los parámetros de

³¹¹ Dong Yu, Shizhen Wang, y Li Deng. Sequential labeling using deep-structured conditional random fields. *IEEE Journal of Selected Topics in Signal Processing*, 4(6):965–973, Dec 2010. ISSN 1932-4553. DOI: 10.1109/JSTSP.2010.2075990

³¹² Karen Simonyan y Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR'2015*, arXiv:1409.1556, 2014. URL <http://arxiv.org/abs/1409.1556>

³¹³ Ruslan Salakhutdinov y Geoffrey Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. En Marina Meila y Xiaotong Shen, editores, *AISTATS'2007 Proceedings of the 11th International Conference on Artificial Intelligence and Statistics*, volume 2 de *Proceedings of Machine Learning Research*, pages 412–419, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR. URL <http://proceedings.mlr.press/v2/salakhutdinov07a.html>

la red (algoritmo *greedy*) y se entrena la capa i mientras se mantienen fijos los parámetros de las demás (por capas). Este pre-entrenamiento se realiza de forma no supervisada para extraer características potencialmente útiles y va seguido de un proceso supervisado de ajuste [*fine-tuning*] de los parámetros de la red. El pre-entrenamiento de la red suele ser eficiente, al emplear una estrategia *greedy*, si bien el ajuste final de los parámetros de la red puede resultar costoso computacionalmente si involucra el ajuste de la red completa obtenida tras el pre-entrenamiento. Es mucho más eficiente si se limita a entrenar una red simple sobre las características extraídas en la fase de pre-entrenamiento.

En definitiva, el pre-entrenamiento se utiliza para aprender acerca de la distribución de los datos de entrada. Ese aprendizaje, usado como etapa previa de un proceso posterior, permite guiar la selección de los parámetros iniciales de una red neuronal profunda, por lo que puede verse el pre-entrenamiento como un mecanismo de inicialización de los parámetros de la red. En el contexto del aprendizaje supervisado, el pre-entrenamiento puede verse como un mecanismo de regularización, que permite a la red generalizar mejor (disminuir su error en el conjunto de prueba) o, al menos, una forma de reducir la varianza. La inicialización de los parámetros de la red los sitúa en una región adecuada del espacio, de la que no escapan cuando se utiliza un algoritmo de optimización basado en el gradiente descendente estocástico, con lo que los resultados obtenidos al entrenar una red neuronal pre-entrenada suelen ser más consistentes que si no se utiliza este tipo de inicialización de sus parámetros.^{314,315}

Como ya vimos al analizar las estrategias de inicialización de los pesos de una red neuronal, la selección inicial de los parámetros de la red puede tener un gran impacto en el rendimiento de la red entrenada. Además de su efecto regularizador, de reducción de la varianza del modelo, el pre-entrenamiento puede mejorar los resultados que se podrían obtener sin él. Es una de las técnicas que contribuyó a la popularización del *deep learning*.

Su principal desventaja es que requiere la realización de dos procesos de entrenamiento, el de pre-entrenamiento inicial y el de entrenamiento de la red final, cada una con su conjunto de hiperparámetros que tendremos que ajustar. Por ejemplo, podemos utilizar el error en el conjunto de validación de la fase final de aprendizaje supervisado para seleccionar los hiperparámetros de la fase de pre-entrenamiento. Sin embargo, el ajuste por separado de los hiperparámetros de los dos procesos de entrenamiento puede resultar muy costoso. Una alternativa más económica computacionalmente consiste en combinar ambas fases en un proceso guiado por un único hiperparámetro, en forma de coeficiente asociado a la función de coste del proceso de pre-entrenamiento. Este coeficiente se utiliza para que el pre-entrenamiento regularice el entrenamiento de la red final, regularización que se puede reducir reduciendo su valor.³¹⁶

³¹⁴ Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, y Pascal Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. En David van Dyk y Max Welling, editores, *AISTATS'2009 Proceedings of the 12th International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 153–160, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR. URL <http://proceedings.mlr.press/v5/erhan09a.html>

³¹⁵ Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, y Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11: 625–660, March 2010. ISSN 1532-4435. URL <http://www.jmlr.org/papers/v11/erhan10a.html>

³¹⁶ Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, y Pascal Lamblin. Exploring Strategies for Training Deep Neural Networks. *Journal of Machine Learning Research*, 10:1–40, June 2009. ISSN 1532-4435. URL <http://www.jmlr.org/papers/v10/larochelle09a.html>

Actualmente, el uso de técnicas de pre-entrenamiento ha decaído en la práctica, sustituidas por modelos regularizados con *dropout* o normalización por lotes, que alcanzan un rendimiento similar al de los seres humanos en tareas que, hasta hace pocos años, se hallaban fuera del alcance de las técnicas de Inteligencia Artificial, si bien para lograrlo suelen necesitar enormes conjuntos de datos de entrenamiento.

El uso de un pre-entrenamiento supervisado *greedy* por capas no suele resultar necesario hoy en día. El no supervisado puede seguir siendo útil. ¿Cuándo? Cuando el número de ejemplos etiquetados es muy pequeño. En ese caso, los datos no etiquetados nos aportarán información adicional que podemos aprovechar y, cuantos más datos tengamos, mejor rendimiento podemos esperar de obtener. Esto puede resultar particularmente útil para resolver problemas de aprendizaje por transferencia o en situaciones en las que la representación de los datos no resulta demasiado informativa. Por ejemplo, se sigue empleando en procesamiento del lenguaje natural para representar palabras en forma de vectores de características [*word embeddings*] que permitan codificar la similitud entre palabras diferentes, representación que se obtiene a partir de conjuntos de textos con billones de palabras.

El aprendizaje por transferencia [*transfer learning*] implica reutilizar características extraídas para resolver un problema en la resolución de un problema diferente, motivo por el que puede recibir el nombre de aprendizaje multitarea [*multitask learning*] cuando se resuelven tareas de distinto tipo o adaptación del dominio [*domain adaptation*] cuando resuelve el mismo tipo de problema en un contexto diferente. En el primer caso, se suelen reutilizar características a bajo nivel y se entranan las capas superiores de la red. Por ejemplo, para reconocer objetos en imágenes, se pueden reutilizar las características extraídas del análisis de imágenes naturales. En el segundo caso, se reutilizan características de alto nivel y se entranan las capas inferiores de la red para adaptarlas a situaciones específicas. Por ejemplo, en un sistema de reconocimiento de voz, se puede adaptar el sistema a las peculiaridades de una voz particular o al tipo de ruido con el que se recibe la señal en función del micrófono y del entorno en el que se emplea el sistema.

También se puede utilizar aprendizaje por transferencia en situaciones en las que se producen cambios graduales con el tiempo, un fenómeno conocido en aprendizaje automático como *concept drift*, que se podría traducir literalmente como deriva de conceptos. Este tipo de cambios graduales pueden observarse, por ejemplo, en sistemas de monitorización como los sistemas de detección de intrusiones o los sistemas de prevención de fraudes. Formalmente, son ejemplos de sistemas de detección de anomalías, con la peculiaridad de que las anomalías van cambiando conforme los atacantes adaptan sus estrategias a las capacidades de detección del sistema, en una carrera de armamento sin fin, un escenario

coloquialmente denominado “carrera de ratas” [*rat race*].

En casos extremos, el aprendizaje por transferencia se puede aplicar en situaciones en las que disponemos de muy pocos o incluso de ningún ejemplo etiquetado. Es el caso del aprendizaje en un intento [*one-shot learning*] o del aprendizaje sin datos [*zero-data* o *zero-shot learning*]:

- En el aprendizaje en un intento [*one-shot learning*], sólo disponemos de un ejemplo etiquetado para resolver el problema de aprendizaje por transferencia. En visión artificial, puede servirnos para extraer toda la información que podamos de una única imagen de un tipo de objeto que previamente desconocíamos.³¹⁷ En procesamiento de lenguaje natural, puede servirnos para completar una frase en la que falta una palabra.³¹⁸
- El aprendizaje sin datos [*zero-shot learning*] puede parecer paradójico. En realidad, es posible si somos capaces de explotar información adicional obtenida en el aprendizaje de otros modelos. Resulta necesario cuando un modelo debe ser capaz de generalizar correctamente a la hora de identificar nuevas clases para las que no se dispone de ejemplos en el conjunto de entrenamiento.³¹⁹ Es lo que nos permite, en la vida real, reconocer un objeto que no hayamos visto nunca en una imagen cuando disponemos de una descripción textual del objeto que sea lo suficientemente precisa. En aprendizaje automático, podemos utilizar la información semántica proveniente de fuentes de datos independientes del conjunto de entrenamiento. Por ejemplo, se ha utilizado para predecir la palabra en la que una persona está pensando a partir de la imagen de una resonancia magnética de su cerebro.³²⁰ De forma mucho más parecida al modo en que nosotros lo utilizamos, el aprendizaje sin datos se ha utilizado para reconocer objetos en imágenes, tanto de clases conocidas como de clases desconocidas, a partir de las similitudes entre palabras que se extraen del análisis de grandes volúmenes de información textual: las imágenes se representan en forma de vectores de características [*image embeddings*] que se pueden comparar con una representación vectorial de palabras [*word embeddings*].³²¹ Es la misma idea que permite que el traductor neuronal de Google, estrenado a finales de 2016 en sustitución de su traductor estadístico previo, pueda traducir textos entre pares de idiomas para los que no ha sido explícitamente entrenado.³²²

Aun cuando nosotros no pre-entrenamos nuestros propios modelos, podemos aprovechar el pre-entrenamiento, supervisado o no, que otros hayan realizado para aprovechar las ventajas que ofrece el aprendizaje por transferencia.³²³

Muchos investigadores publican los parámetros de sus redes entrenadas para facilitar que otros las podamos utilizar de punto de partida en nuestras propias aplicaciones. Por ejemplo, en Internet se pueden

³¹⁷ Li Fei-Fei, R. Fergus, y P. Perona. One-shot learning of object categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):594–611, April 2006. ISSN 0162-8828. DOI: 10.1109/TPAMI.2006.79

³¹⁸ Oriol Vinyals, Charles Blundell, Timothy P. Lillicrap, Koray Kavukcuoglu, y Daan Wierstra. Matching Networks for One Shot Learning. *arXiv e-prints*, arXiv:1606.04080, 2016. URL <http://arxiv.org/abs/1606.04080>

³¹⁹ Hugo Larochelle, Dumitru Erhan, y Yoshua Bengio. Zero-data Learning of New Tasks. En *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 2 of *AAAI'08*, pages 646–651. AAAI Press, 2008. ISBN 978-1-57735-368-3. URL <http://dl.acm.org/citation.cfm?id=1620163.1620172>

³²⁰ Mark Palatucci, Dean Pomerleau, Geoffrey E Hinton, y Tom M. Mitchell. Zero-shot learning with semantic output codes. En *NIPS'2009 Advances in Neural Information Processing Systems 22*, pages 1410–1418. Curran Associates, Inc., 2009. URL <https://goo.gl/K7uCFh>

³²¹ Richard Socher, Milind Ganjoo, Christopher D. Manning, y Andrew Y. Ng. Zero-shot learning through cross-modal transfer. En *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, pages 935–943, USA, 2013. Curran Associates Inc. URL <https://arxiv.org/abs/1301.3666>

³²² Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda B. Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, y Jeffrey Dean. Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation. *arXiv e-prints*, abs/1611.04558, 2016b. URL <http://arxiv.org/abs/1611.04558>

³²³ Jason Yosinski, Jeff Clune, Yoshua Bengio, y Hod Lipson. How transferable are features in deep neural networks? En *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3320–3328. MIT Press, 2014. URL <https://goo.gl/AZacCB>

descargar ficheros con los parámetros de redes entrenadas para reconocer objetos en imágenes, entrenadas utilizando la base de datos ImageNet (<http://www.image-net.org/>) y que han llegado a ganar competiciones internacionales como ILSVRC [*ImageNet Large Scale Visual Recognition Competition*].³²⁴ También se pueden conseguir los vectores pre-entrenados con los que representar palabras, útiles en la resolución de múltiples problemas de procesamiento del lenguaje natural,³²⁵ como es el caso paradigmático de `word2vec`.³²⁶

Tasas de aprendizaje

Una vez inicializados correctamente los parámetros de nuestra red neuronal artificial, tenemos que ajustarlos utilizando un algoritmo de entrenamiento. Si usamos el gradiente descendente, la actualización de los pesos de la red se hará de acuerdo a una expresión de la forma

$$\Delta w = -\eta \nabla E$$

Esta expresión se acaba traduciendo en la siguiente fórmula de actualización de los pesos de la red

$$\Delta w_{ij} = -\eta x_i \delta_j$$

donde δ_j representa la derivada parcial del error con respecto a la entrada neta z_j de la neurona j de una capa con entradas x_i . La magnitud del ajuste de los pesos dependerá, pues, de la magnitud del gradiente de la señal de error, de las entradas recibidas y de un parámetro del algoritmo, η , su tasa de aprendizaje.

La regla de ajuste de los pesos se basa en la estimación de un gradiente. Dicha estimación es numéricamente inestable, como suele ser el de cualquier derivada calculada numéricamente, y los errores de aproximación se van multiplicando, lo que puede ocasionar problemas de convergencia en el algoritmo de optimización de los parámetros de la red, basado en el gradiente descendente.

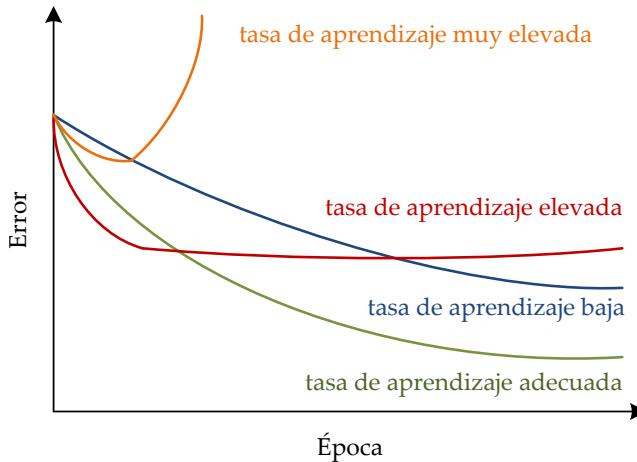
En una red multicapa, los gradientes del error que se propagan pueden crecer (explotar) o desvanecerse (desaparecer) relativamente rápido, lo que dificulta el proceso de optimización de los parámetros de la red. La tasa de aprendizaje es el parámetro clave cuyo ajuste puede ayudarnos a conseguir un funcionamiento adecuado del proceso de entrenamiento de la red, ya que determina cuánto se ajustan los pesos en cada iteración del algoritmo. Para muchos, se trata del hiperparámetro más importante del algoritmo de entrenamiento de una red.

Como criterio heurístico general, buscaremos elegir un valor para la tasa de aprendizaje η que haga que, en cada iteración, los pesos cambien en una fracción de su valor actual. Suele ser útil comparar la magnitud

³²⁴ Maxime Oquab, Leon Bottou, Ivan Laptev, y Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. En *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1717–1724, June 2014. doi: 10.1109/CVPR.2014.222

³²⁵ Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, y Pavel Kuksa. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, 12:2493–2537, November 2011b. ISSN 1532-4435. URL <http://www.jmlr.org/papers/v12/colllobert11a.html>

³²⁶ Tomas Mikolov, Kai Chen, Greg Corrado, y Jeffrey Dean. Efficient estimation of word representations in vector space. *ICLR'2013*, arXiv:1301.3781, 2013a. URL <http://arxiv.org/abs/1301.3781>



de los gradientes del error con la magnitud de los parámetros que pretendemos ajustar. Normalmente, nos gustaría que las actualizaciones de los parámetros correspondiesen a un cambio más cerca del 1% del valor del parámetro que al 50% o al 0.001%.³²⁷

El criterio anterior, abiertamente ambiguo, pretende llegar a un equilibrio que facilite el entrenamiento de la red. Si la tasa de aprendizaje es demasiado pequeña, la convergencia del algoritmo de entrenamiento de la red puede ser innecesariamente lento al ir modificando los pesos muy lentamente. Pero, si es demasiado elevada, puede desencadenar oscilaciones y dar lugar a un comportamiento inestable del proceso de aprendizaje, con lo que el error no resultaría aceptable.

Si en el proceso de preparación de los datos (esto es, preprocessamiento de las entradas x_i) o en la estrategia de inicialización de la red (que puede influir en la saturación de las neuronas y, por ende, en la magnitud de los gradientes δ_j), hemos tomado medidas para fomentar la estabilidad del algoritmo de aprendizaje, fijar una tasa de aprendizaje adecuada será relativamente más sencillo que en ausencia de tales medidas.

De hecho, muchas de las reglas de tipo heurístico que empleamos para diseñar un mecanismo adecuado de inicialización de los pesos de la red las podemos aplicar también a las tasas de aprendizaje:

- La tasa de aprendizaje η tendrá un valor pequeño inicialmente, aunque el valor adecuado dependerá del problema particular y de otros hiperparámetros del algoritmo (v.g., si utilizamos momentos o no en las actualizaciones de los pesos). Podemos comenzar con un valor inicial $\eta = 0.1$ y explorar los efectos que se producen al doblar y dividir por dos la tasa de aprendizaje.
- Las neuronas con muchas entradas deberían tener una tasa de aprendizaje menor que las neuronas con pocas entradas, para que la velocidad del proceso de aprendizaje sea similar en todas las capas de la red.³²⁸

Figura 130: Representación gráfica del efecto de la tasa de aprendizaje sobre la convergencia del algoritmo de entrenamiento de una red neuronal multicapa. Adaptada de <http://cs231n.github.io/neural-networks-3/#loss>.

³²⁷ Léon Bottou. Multilayer Neural Networks. *Deep Learning Summer School, Montreal, 2012*. URL http://videolectures.net/deeplearning2015_bottou_neural_networks/

³²⁸ Gerald Tesuaro y Bob Janssens. Scaling Relationships in Back-propagation Learning. *Complex Systems*, 2(1): 39–44, 1988. ISSN 0891-2513. URL http://www.complex-systems.com/abstracts/v02_i01_a03.html

La recomendación, idéntica a la que ya hicimos para inicializar los pesos de la red, consiste en hacer que la tasa de aprendizaje sea inversamente proporcional al número de entradas de las neuronas de cada capa: η_0 / \sqrt{n} . Obviamente, si este criterio ya lo hemos utilizado al inicializar los pesos de la red, no es necesario que lo volvamos a aplicar aquí para establecer la tasa de aprendizaje.

- Todas las neuronas de una red multicapa, idealmente, deberían aprender al mismo ritmo. Dado que el gradiente del error suele ser mayor en las capas más cercanas a la capa de salida de la red que en las capas más cercanas a la entrada, la tasa de aprendizaje debería tomar valores menores en las capas posteriores de la red. De nuevo, si en la inicialización de los pesos ya hemos tomado medidas para ecualizar el gradiente del error a lo largo de la red (v.g. inicialización de Xavier), tampoco sería necesario establecer tasas de aprendizaje diferentes para las distintas capas.

Para igualar la velocidad del aprendizaje, además de hacer que las tasas de aprendizaje dependan del número de entradas de cada neurona o de la capa de la red en la que se encuentre, podemos establecer alguna estrategia que permita adaptar las tasas de aprendizaje. Esto es, en vez de establecer una tasa global de aprendizaje, fija para toda la red o común para todas las neuronas de una misma capa, podemos diseñar heurísticas que vayan adaptando la tasa de aprendizaje. Las tasas de aprendizaje adaptables pueden establecerse a nivel global o localmente, con tasas específicas para cada capa, para cada neurona o, incluso, para cada peso concreto de la red.

Una posibilidad, común en la literatura sobre aproximación estocástica, consiste simplemente en hacer que la tasa de aprendizaje vaya disminuyendo progresivamente.³²⁹ Al principio, se emplea una tasa de aprendizaje más elevada para acelerar la reducción del error durante las primeras iteraciones del algoritmo. A continuación, se va reduciendo la tasa de aprendizaje conforme queremos afinar más en los valores de los parámetros de la red, reduciendo las oscilaciones causadas por una tasa de aprendizaje demasiado elevada. La estrategia más utilizada consiste en establecer la tasa de aprendizaje de acuerdo a

$$\eta(t) = \frac{\kappa}{t}$$

donde κ es una constante que permite garantizar la convergencia en problemas de optimización convexa si es lo suficientemente pequeña.³³⁰

El principal inconveniente del ajuste anterior de la tasa de aprendizaje es que decrece demasiado rápidamente al principio. Una alternativa habitual consiste en emplear el algoritmo de búsqueda y convergencia [*search-then-converge schedule*] propuesto por Darken y Moody:

$$\eta(t) = \frac{\eta_0}{1 + t/\tau}$$

³²⁹ Herbert Robbins y Sutton Monroe. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951. ISSN 0003-4851. URL <http://www.jstor.org/stable/2236626>

³³⁰ Lennart Ljung. Analysis of recursive stochastic algorithms. *IEEE Transactions on Automatic Control*, 22(4):551–575, Aug 1977. ISSN 0018-9286. DOI: 10.1109/TAC.1977.1101561

donde η_0 es la tasa de aprendizaje inicial y $\tau \gg 1$ es una constante de tiempo (“el tiempo de búsqueda”, p.ej. $100 \leq \tau \leq 500$).

El proceso de búsqueda y convergencia consta de dos fases:^{331,332}

- En la fase de búsqueda, durante las iteraciones iniciales del algoritmo, el valor de t es pequeño en comparación con la constante de tiempo τ . La tasa de aprendizaje decrece lentamente y es casi constante: $\eta(t) \approx \eta_0$. El algoritmo se comporta como si estuviésemos utilizando una tasa de aprendizaje fija.
- En la fase de convergencia, cuando el valor de t es grande en comparación con la constante de tiempo τ , la tasa de aprendizaje decrece como en la aproximación estocástica tradicional, de Robbins y Monroe, con $\kappa = \tau\eta_0$.

El proceso de adaptación de la tasa de aprendizaje es similar al enfriamiento simulado. Al comienzo del aprendizaje, el algoritmo escapa de mínimos locales poco profundos al usar una tasa de aprendizaje η_0 relativamente alta, con la que esperamos llegar a una zona prometedora del espacio de búsqueda. A continuación, el proceso de reducción de la tasa de aprendizaje hace que converjamos a un mínimo dentro de esa zona prometedora a la que llegamos en la fase de búsqueda. El objetivo del algoritmo de búsqueda y convergencia es evitar que nos quedemos atrapados en mínimos locales al principio y garantizar que encontramos un óptimo al final, intentando combinar las mejores características de una tasa de aprendizaje fija y de una aproximación estocástica.

Los métodos anteriores son sólo algunas de las posibilidades que tenemos a la hora de ajustar dinámicamente la tasa de aprendizaje conforme avanza el proceso de entrenamiento de la red. Es habitual, en la práctica, que la tasa de aprendizaje se reduzca linealmente:

$$\eta(t) = (1 - \alpha(t))\eta_0 + \alpha(t)\eta_\tau$$

con $\alpha(t) = t/\tau$, lo que hace que la tasa de aprendizaje $\eta(t)$ decrezca linealmente hasta la iteración τ . A partir de esa iteración, lo más común es dejar que la tasa de aprendizaje se mantenga constante (η_τ).

Cuando se emplea aprendizaje por lotes, se pueden emplear otras heurísticas para ajustar la tasa de aprendizaje a nivel global. El “conductor audaz” [*bold-driver*]^{333,334} adapta la tasa de aprendizaje monitorizando la señal de error tras cada cambio en los pesos de la red. Usualmente, el número de iteraciones necesarias para que el algoritmo converja desciende conforme aumenta la tasa de aprendizaje pero, a partir de cierto valor de ésta, el algoritmo comienza a oscilar. La siguiente heurística intenta adaptar dinámicamente la tasa de aprendizaje para que se mantenga lo más cerca posible de su valor máximo sin provocar inestabilidad en el

³³¹ Christian Darken y John Moody. Note on learning rate schedules for stochastic optimization. En *Proceedings of the 3rd International Conference on Advances in Neural Information Processing Systems*, NIPS’90, pages 832–838. Morgan Kaufmann Publishers Inc., 1990. ISBN 1-55860-184-8. URL <https://goo.gl/HuC5Xu>

³³² Christian Darken y John Moody. Towards faster stochastic gradient search. En *Proceedings of the 4th International Conference on Neural Information Processing Systems*, NIPS’91, pages 1009–1016. Morgan Kaufmann Publishers Inc., 1991. ISBN 1-55860-222-4. URL <https://goo.gl/1Ypojb>

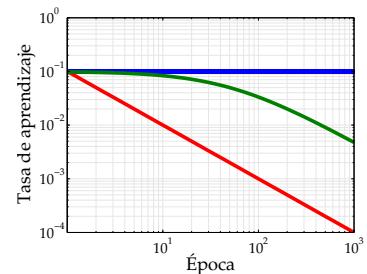


Figura 131: Comparación de distintas formas de ajustar la tasa de aprendizaje durante el entrenamiento de una red neuronal: tasa de aprendizaje fija (azul), aproximación estocástica (rojo), búsqueda y convergencia (verde). Tenga en cuenta que la gráfica se muestra con ambos ejes en escala logarítmica

³³³ Roberto Battiti. Accelerated Backpropagation Learning: Two Optimization Methods. *Complex Systems*, 3(4):331–342, 1980. ISSN 0891-2513. URL http://www.complex-systems.com/abstracts/v03_i04_a02.html

³³⁴ T. P. Vogl, J. K. Mangis, A. K. Ringer, W. T. Zink, y D. L. Alkon. Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, 59(4):257–263, Sep 1988. ISSN 1432-0770. DOI: 10.1007/BF00332914

proceso de aprendizaje:

$$\eta(t+1) = \begin{cases} \rho^+ \eta(t) & \text{si } \Delta E(t) < 0 \\ \rho^- \eta(t) & \text{si } \Delta E(t) > 0 \end{cases}$$

donde ρ^+ es una constante ligeramente mayor que 1 (típicamente, 1.1), ρ^- es una constante significativamente menor que 1 (típicamente, 0.5) y $\Delta E(t)$ es la variación del error observado $\Delta E(t) = E(t) - E(t-1)$.

Cuando el error disminuye, $\Delta E(t) < 0$, nos estamos acercando a un mínimo y podemos aumentar la tasa de aprendizaje η para acelerar la convergencia. Cuando aumenta, $\Delta E(t) > 0$, nos hemos pasado del mínimo y conviene reducir la tasa de aprendizaje. En la propuesta original, los pesos no se actualizan en este caso, sólo cuando disminuye el error.

Otra posible heurística de adaptación de la tasa de aprendizaje se basa en la regla delta de Widrow y Hoff aplicando el principio de mínima perturbación (adaptar los pesos para reducir el error perturbando lo menos posible las salidas ya aprendidas). En este caso, se ajusta la tasa de aprendizaje en función de las entradas $x(t)$ en cada momento:³³⁵

$$\eta(t) = \frac{\eta_0}{\|x(t)\|_2^2}$$

donde η_0 es la tasa de aprendizaje para entradas ya normalizadas, usualmente $0.1 \leq \eta_0 \leq 1$.

Si, en vez de usar la entrada $x(t)$, utilizamos el gradiente del error $E(t)$, obtenemos el ajuste de la tasa de aprendizaje utilizada por el *backpropagation* adaptable [ABP: *Adaptive BackPropagation*]:³³⁶

$$\eta(t) = \eta_0 \frac{\rho(E(t))}{\|\nabla_w E(t)\|}$$

donde $\rho(E)$ es una función del error, típicamente $\rho(E) = E$. Aunque la adaptación de la tasa de aprendizaje acelera la convergencia del algoritmo, su principal inconveniente es que la tasa de aprendizaje $\eta(t)$ se dispara cuando estamos cerca de un punto crítico de la función de error, ya que $\|\nabla_w E(t)\| \approx 0$ en el entorno de un mínimo local.

Se pueden utilizar otros criterios para ajustar la tasa de aprendizaje a nivel global, modificando el tamaño del paso que se realiza en cada iteración del gradiente descendente. El algoritmo *backpropagation* con paso de tamaño variable [BPSV: *BP with Variable StepSize*]³³⁷ aproxima la constante de Lipschitz como

$$L(t) = \frac{\|\nabla_w E(t) - \nabla_w E(t-1)\|}{\|w(t) - w(t-1)\|}$$

para utilizarla en la definición de la tasa de aprendizaje del algoritmo:

$$\eta(t) = \frac{1}{2} L(t)^{-1} = \frac{1}{2} \frac{\|w(t) - w(t-1)\|}{\|\nabla_w E(t) - \nabla_w E(t-1)\|}$$

³³⁵ Bernard Widrow y Michael A. Lehr. 30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, Sep 1990. ISSN 0018-9219. DOI: 10.1109/5.58323

³³⁶ A. G. Parlos, B. Fernandez, A. F. Atiya, J. Muthusami, y W. K. Tsai. An accelerated learning algorithm for multi-layer perceptron networks. *IEEE Transactions on Neural Networks*, 5(3):493–497, May 1994. ISSN 1045-9227. DOI: 10.1109/72.286921

³³⁷ George D. Magoulas, Michael N. Vrahatis, y George S. Androutsakis. Effective backpropagation training with variable stepsize. *Neural Networks*, 10(1):69 – 82, 1997. ISSN 0893-6080. DOI: 10.1016/S0893-6080(96)00052-4

Se dice que una función $f(x)$ es una función de Lipschitz si verifica $|f(x) - f(y)| \leq L|x - y|$, donde L es una constante independiente de x e y

Si $\eta(t)$ queda por debajo de un valor mínimo η_{min} , cota inferior del tamaño del paso, entonces se multiplica por 2^{k-1} , donde k es el número de épocas en las que lleva sin satisfacer la condición de Armijo:

$$E(w - \eta \nabla E) - E(w) \leq -\frac{1}{2} \eta \|\nabla E\|^2$$

La heurística anterior está diseñada pensando en un entrenamiento por lotes de la red neuronal. También se puede estimar la tasa de aprendizaje óptima cuando utilizamos el gradiente descendente estocástico. Para ello, se puede estimar el *eigenvector* principal de la matriz Hessiana (la segunda derivada de la función objetivo) sin necesidad de calcular dicha matriz.³³⁸ El algoritmo sería el siguiente:

- Se escogen un vector aleatorio Ψ y dos constantes positivas pequeñas, α y γ (p.ej. $\alpha = \gamma = 0.01$).
- Se elige un ejemplo del conjunto de entrenamiento, x , para el que se calcula el gradiente del error asociado al ejemplo de la forma tradicional, $G_x = \nabla_w E(w)$, y el gradiente del error utilizando una versión perturbada del vector de pesos $G_p = \nabla_w E(w + \alpha N(\Psi))$, donde $N(\Psi)$ es la versión normalizada del vector Ψ , $N(\Psi) = \Psi / \|\Psi\|$.
- Se actualiza el vector Ψ usando una media móvil: $\Psi \leftarrow (1 - \gamma)\Psi + \gamma/\alpha(G_p - G_x)$
- Se restauran los pesos originales w y se repite el proceso iterativamente hasta que $\|\Psi\|$ se estabilice.
- La tasa de aprendizaje óptima es $\|\Psi\|^{-1}$. ¿Por qué? Porque es la que nos permitiría saltar al mínimo en un solo paso en una función cuadrática unidimensional. En el caso multidimensional, el análisis es algo más complicado, pero la inversa del mayor *eigenvalue* de la matriz Hessiana nos da un valor que asegura la convergencia del algoritmo y está cerca del óptimo (para funciones cuadráticas, que nos sirven como aproximaciones locales válidas de la función de error real).

Usemos la heurística que usemos, el ajuste de la tasa de aprendizaje siempre sigue el mismo esquema general:

- Comenzamos con una estimación inicial de la tasa de aprendizaje.
- Si notamos que el error se va reduciendo de forma consistente pero demasiado lenta, entonces aumentamos la tasa de aprendizaje para intentar acelerar la convergencia del algoritmo.

Incrementar la tasa de aprendizaje conforme el error decrece puede ayudar a acelerar la convergencia del algoritmo hacia el mínimo de la función de error, si bien nos podemos pasar y el error puede comenzar a oscilar. Si la tasa de aprendizaje es demasiado alta, el algoritmo es inestable y diverge.

La condición de Armijo comprueba si un paso de la posición actual x a una posición modificada $x + \delta$ consigue una disminución adecuada en la función objetivo que pretendemos minimizar.

³³⁸ Yann LeCun, Patrice Y. Simard, y Barak Pearlmuter. Automatic Learning Rate Maximization by On-Line Estimation of the Hessian's Eigenvectors. En S. J. Hanson, J. D. Cowan, y C. L. Giles, editores, *Advances in Neural Information Processing Systems 5*, pages 156–163. Morgan-Kaufmann, 1993. URL <https://goo.gl/TW7uCr>

El parámetro α controla el tamaño de la perturbación. Cuanto menor sea, mejor será la estimación pero más probable es la aparición de errores numéricos. El parámetro γ equilibra la velocidad de convergencia de Ψ y la precisión del resultado.

- Si observamos que el error crece u oscila, pues, debemos reducir de forma automática la tasa de aprendizaje.

Si utilizamos una estimación estocástica del gradiente, ya sea con aprendizaje *online* o usando aprendizaje con mini-lotes, casi siempre suele ayudar que reduzcamos la tasa de aprendizaje al final del entrenamiento. De esta forma, eliminamos posibles fluctuaciones en los pesos finales debidas a variaciones entre mini-lotes (o muestras individuales, en el caso del aprendizaje *online*).

También podemos recurrir a un conjunto de validación diferente al conjunto de datos de entrenamiento. Usando una estimación del error sobre el conjunto de validación, reduciremos la tasa de aprendizaje cuando la estimación del error cometido deje de disminuir.

Ahora bien, debemos ser cuidadosos para no ser demasiado agresivos a la hora de reducir la tasa de aprendizaje. Reducir la tasa de aprendizaje reduce las fluctuaciones aleatorias debidas a las estimaciones del gradiente del error de los distintos mini-lotes, pero hace el aprendizaje más lento. Si reducimos la tasa de aprendizaje demasiado pronto, puede que los resultados que obtengamos sean subóptimos.

Un problema frecuente es que, si utilizamos, de inicio, una tasa de aprendizaje demasiado alta, los pesos correspondientes a las neuronas ocultas de la red adquirirán valores extremos (valores positivos muy altos o valores negativos muy bajos). Esto fomentará que las neuronas se saturen, con lo que el gradiente del error en las capas ocultas de la red será minúsculo. El problema de la desaparición del gradiente hará que seamos incapaces de corregir adecuadamente los valores de los pesos y el error de la red no disminuirá. La saturación de las neuronas ocultas hace que nos quedemos estancados en una meseta [*plateau*], que resulta fácil de confundir con un mínimo local.

La función de error de una red neuronal multicapa puede que no sólo tenga zonas muy planas, correspondientes a mesetas en las que el gradiente es prácticamente cero, sino que también puede tener “precipicios” o “acantilados” en los que la función de error fluctúa de forma brusca. En esas zonas, el gradiente de la función de error se dispara y, si lo utilizamos directamente, puede que saltemos demasiado al actualizar los pesos de la red. Para evitar ese tipo de situaciones, se recomienda “recortar el gradiente” [*gradient clipping*]: limitar los valores permitidos para el gradiente antes de aplicar la fórmula de actualización de los pesos. Al fin y al cabo, el gradiente no indica el tamaño óptimo del paso, sino la dirección en la que hay que darlo. El tamaño del paso debería establecerlo la tasa de aprendizaje. Una de las formas de recortar el gradiente es limitar su magnitud, de forma que si $\|\nabla E\| > v$, entonces $\nabla E \leftarrow v \nabla E / \|\nabla E\|$.³³⁹

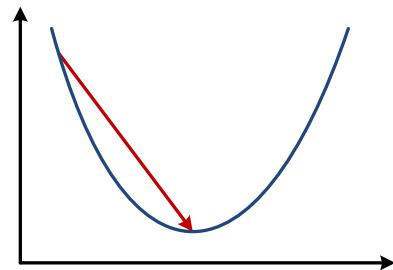


Figura 132: Convergencia del gradiente descendente: La tasa de aprendizaje óptima η^* nos lleva al mínimo de la función de error en un solo paso.

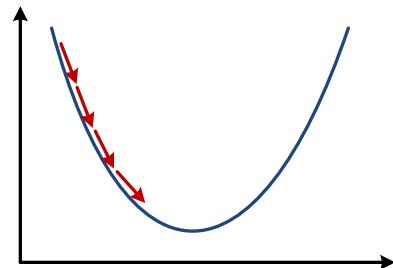


Figura 133: Convergencia del gradiente descendente: Una tasa de aprendizaje pequeña, $\eta < \eta^*$, ralentiza la convergencia del algoritmo.

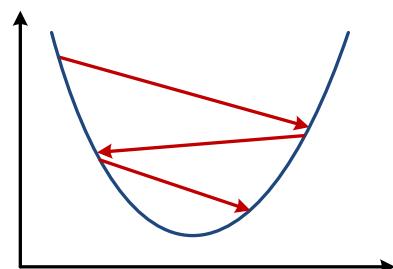


Figura 134: Convergencia del gradiente descendente: Una tasa de aprendizaje grande, $\eta > \eta^*$, provoca oscilaciones.

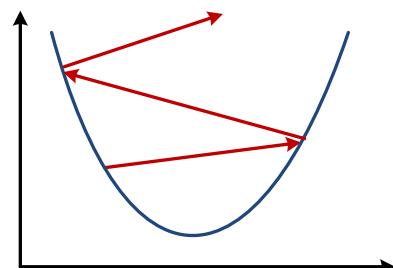


Figura 135: Convergencia del gradiente descendente: Una tasa de aprendizaje demasiado grande causa la divergencia del algoritmo. En el caso idealizado de una función cuadrática, esto ocurre cuando $\eta > 2\eta^*$.

³³⁹ Razvan Pascanu, Tomas Mikolov, y Yoshua Bengio. On the difficulty of training recurrent neural networks. En *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III.1310–1318. JMLR.org, 2013. URL <http://proceedings.mlr.press/v28/pascanu13.pdf>

Recortar el gradiente es una de las estrategias que se pueden emplear para mitigar el problema de la explosión del gradiente, opuesto a la desaparición del gradiente, y que también resulta problemático al entrenar redes neuronales artificiales. Otra posibilidad, como ya vimos, consiste en inicializar los pesos de la red de forma que se prevenga este tipo de problemas.³⁴⁰

Tasas locales de aprendizaje

Los esquemas de adaptación de las tasas de aprendizaje que hemos visto hasta ahora utilizan criterios globales para establecer un valor adecuado para las tasas de aprendizaje utilizadas en el gradiente descendente. Sin embargo, también podemos diseñar estrategias en las que cada conexión sináptica de la red tenga su propia tasa de aprendizaje. Tendremos, entonces, una tasa de aprendizaje para cada peso de la red:

$$\Delta w_{ij}^k = -\eta_{ij}^k \frac{\partial E}{\partial w_{ij}^k} = -\eta_{ij}^k \delta_{ij}^k$$

donde el superíndice k indica la capa de la red, i la entrada y j la salida. Para simplificar algo la notación, prescindiremos de los superíndices en el resto de la sección.

Veamos algunas de las muchas heurísticas³⁴¹ que se han propuesto para ajustar las tasas locales de aprendizaje:

- La regla delta-barra-delta [*delta-bar-delta*],³⁴² propuesta por Robert Jacobs, de la Universidad de Massachusetts, es una de las heurísticas más conocidas. Consiste en aumentar la tasa de aprendizaje asociada a un parámetro de la red cuando se mantiene el signo de la derivada parcial del error con respecto a ese parámetro (esto es, se han realizado múltiples actualizaciones seguidas del parámetro en el mismo sentido). Si se detectan cambios en el signo de la derivada (cambios en el sentido de ajuste del parámetro correspondientes a una oscilación), la tasa de aprendizaje se reduce.

En primer lugar, calculamos la derivada del error con respecto a cada peso de la red, δ_{ij} :

$$\delta_{ij}(t) = \frac{\partial E}{\partial w_{ij}}(t)$$

A continuación, combinamos las dos últimas derivadas del error para obtener una media móvil $\overline{\delta_{ij}}$ (barra-delta):

$$\overline{\delta_{ij}}(t) = (1 - \beta)\delta_{ij}(t) + \beta\overline{\delta_{ij}}(t - 1)$$

usando un parámetro β entre 0 y 1, p.ej. $\beta = 0.7$.

³⁴⁰ David Sussillo. Random walk initialization for training very deep feed-forward networks. *arXiv e-prints*, arXiv:1412.6558, 2014. URL <http://arxiv.org/abs/1412.6558>

³⁴¹ Sung-Bae Cho y Jin H. Kim. Rapid backpropagation learning algorithms. *Circuits, Systems and Signal Processing*, 12(2):155–175, Jun 1993. ISSN 1531-5878. doi: 10.1007/BF01189872

³⁴² Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295 – 307, 1988. ISSN 0893-6080. doi: 10.1016/0893-6080(88)90003-2

Por último, se comparan las dos últimas medias móviles $\overline{\delta_{ij}}$ (de ahí lo de delta-barra-delta):

$$\eta_{ij}(t) = \begin{cases} \eta_{ij}(t-1) + \kappa & \text{si } \overline{\delta_{ij}}(t) \overline{\delta_{ij}}(t-1) > 0 \\ \eta_{ij}(t-1) * (1 - \gamma) & \text{si } \overline{\delta_{ij}}(t) \overline{\delta_{ij}}(t-1) < 0 \\ \eta_{ij}(t-1) & \text{en otro caso} \end{cases}$$

con parámetros κ y γ , p.ej. $\kappa = 0.035$ y $\gamma = 1/3$.

Es decir, la tasa de aprendizaje local η_{ij} aumenta una constante κ si no hay cambios de signo en las derivadas del error y disminuye en un porcentaje γ de su valor si hay cambios de signo.

- La heurística usada por el método SuperSAB³⁴³ es similar a SA90.³⁴⁴ Ambas actualizan las tasas de aprendizaje locales de acuerdo a una expresión de la forma:

$$\eta_{ij}(t) = \begin{cases} \eta_0^+ \eta_{ij}(t-1) & \text{si } \delta_{ij}(t) \delta_{ij}(t-1) > 0 \\ \eta_0^- \eta_{ij}(t-1) & \text{si } \delta_{ij}(t) \delta_{ij}(t-1) < 0 \end{cases}$$

donde $\eta_0^+ > 0$ y $0 < \eta_0^- < 1$. En el caso de SuperSAB, $\eta_0^+ \simeq 1/\eta_0^-$. Es decir, las correcciones de las tasas de aprendizaje son siempre multiplicativas: aumentan cuando el gradiente del error no cambia de signo y disminuyen cuando lo hace. Sin necesidad de recurrir a medias móviles como en el esquema delta-barra-delta, pero con el inconveniente de que la tasa de aprendizaje η_{ij} puede crecer o decrecer exponencialmente cuando se realizan múltiples actualizaciones sucesivas en el mismo sentido, lo que ralentiza el proceso de aprendizaje.

- En el curso MOOC sobre redes neuronales de Geoffrey Hinton en Coursera,³⁴⁵ se recomienda una estrategia, similar a la heurística delta-barra-delta, en la que se usa una tasa de aprendizaje global, fijada manualmente, y luego se multiplica por una ganancia local adaptable, que se determina para cada peso de forma empírica:

$$\eta_{ij} = \eta g_{ij}$$

Inicialmente, la ganancia local g_{ij} es 1 para todos los pesos. A continuación, se incrementa la ganancia local cuando el gradiente para un peso no cambia de signo y se disminuye cuando lo hace. Como antes, comprobamos el signo del producto $\delta_{ij}(t) \delta_{ij}(t-1)$:

$$g_{ij}(t) = \begin{cases} g_{ij}(t-1) + \delta & \text{si } \delta_{ij}(t) \delta_{ij}(t-1) > 0 \\ g_{ij}(t-1) * (1 - \delta) & \text{si } \delta_{ij}(t) \delta_{ij}(t-1) < 0 \end{cases}$$

donde $\delta = 0.05$.

A diferencia de SuperSAB, que siempre realiza ajustes multiplicativos, Hinton propone realizar aumentos aditivos y descensos multiplicativos en los factores de ganancia asociados a las tasas de aprendizaje

³⁴³ Tom Tollenaeare. SuperSAB: Fast adaptive back propagation with good scaling properties. *Neural Networks*, 3 (5):561 – 573, 1990. ISSN 0893-6080. DOI: 10.1016/0893-6080(90)90006-7

³⁴⁴ Fernando M. Silva y Luis B. Almeida. Speeding up Backpropagation. En R. Eckmiller, editor, *Advanced Neural Computers - Proceedings of the International Symposium on Neural Networks for Sensory and Motor Systems*, page 151–158. North-Holland, 1990. ISBN 978-0-444-88400-8. DOI: 10.1016/B978-0-444-88400-8.50022-4

³⁴⁵ Geoffrey Hinton, Nitsh Srivastava, y Kevin Swersky. Neural networks for machine learning. *Coursera, MOOC video lectures*, 2012a

locales, como en la regla delta-barra-delta (aunque usando un único parámetro δ en lugar de los dos parámetros, κ y γ , de la regla delta-barra-delta, i.e. asumiendo $\kappa = \gamma$).

¿Por qué se utilizan aumentos aditivos y descensos multiplicativos de esta forma? En primer lugar, porque las ganancias elevadas caen rápidamente en cuanto se producen oscilaciones. Además, si el gradiente es totalmente aleatorio, la ganancia se mantendrá en torno a 1 si sumamos $+\delta$ la mitad de las veces y multiplicamos por $(1 - \delta)$ la otra mitad. De esta forma, se preserva la estabilidad del algoritmo de ajuste de las tasas de aprendizaje.

Hinton sugiere algunos trucos adicionales para mejorar el proceso de ajuste de las tasas de aprendizaje. Por ejemplo, igual que cuando recortamos el gradiente [*gradient clipping*], se pueden limitar las ganancias para que siempre se mantengan en un rango razonable, que puede ser el intervalo $[0.1, 10]$ o el intervalo $[0.01, 100]$.

Cuando se utiliza aprendizaje por lotes o minilotes, se reducen los cambios de signo en el gradiente debidos al error de muestreo, que aparecerán con mucha mayor frecuencia si recurrimos al aprendizaje *online*. Cuanto más grandes sean los minilotes que utilicemos, menos probable es que la causa principal de los cambios de signo en el gradiente se deban al error de muestreo cometido por el gradiente descendente estocástico. De forma que se recomienda el uso de aprendizaje por lotes o, en su caso, aprendizaje por minilotes con minilotes relativamente grandes.

- El método Quickprop^{346,347} cambia ligeramente la fórmula típica de actualización de los pesos de la red:

$$\Delta w_{ij}(t) = \begin{cases} \alpha_{ij}(t) \Delta w_{ij}(t-1) & \text{si } \Delta w_{ij}(t-1) \neq 0 \\ \eta_0 \delta_{ij}(t) & \text{si } \Delta w_{ij}(t-1) = 0 \end{cases}$$

donde

$$\alpha_{ij}(t) = \min \left\{ \frac{\delta_{ij}(t)}{\delta_{ij}(t-1) - \delta_{ij}(t)}, \alpha_{max} \right\}$$

siendo α_{max} típicamente igual a 1.75 y η_0 una tasa de aprendizaje que sólo se utiliza al comienzo del algoritmo ($0.01 \leq \eta_0 \leq 0.6$). El parámetro α_{max} se utiliza para evitar actualizaciones demasiado bruscas, como en el recorte del gradiente, y el uso del gradiente en dos instantes de tiempo consecutivos se emplea como aproximación discreta de la segunda derivada del error.

Como su nombre indica, Quickprop suele converger mucho más rápidamente que la versión convencional del gradiente descendente con *backpropagation*.

- No todas las estrategias que se han propuesto para utilizar tasas locales de aprendizaje son adaptativas. El *backpropagation* con ecualización

³⁴⁶ Scott E. Fahlman. Faster-Learning Variations on Back-Propagation: An Empirical Study. En David S. Touretzky, Geoffrey E. Hinton, y Terrence J. Sejnowski, editores, *Proceedings of the 1988 Connectionist Models Summer School*, pages 38–51. Morgan Kaufmann, 1988. URL <https://www.cs.cmu.edu/~sef/sefPubs.htm>

³⁴⁷ Scott E. Fahlman y Christian Lebiere. The cascade-correlation learning architecture. En D. S. Touretzky, editor, *NIPS'1989 Advances in Neural Information Processing Systems 2*, pages 524–532. Morgan-Kaufmann, 1990. URL <https://goo.gl/4tAUaK>

del error [EEBP, *Equalized Error BackPropagation*]³⁴⁸ determina las magnitudes relativas de las tasas locales de aprendizaje a partir de las propiedades de la red y del conjunto de entrenamiento, de forma similar a como establecen la escala de los pesos algunas técnicas de inicialización de los pesos de la red.

En EEBP, que resulta adecuado para el aprendizaje *online*, se utiliza una tasa de aprendizaje global, η , que luego se ajusta con un factor de ganancia g_{ij} para cada neurona particular:

$$\eta_{ij} = \eta g_{ij}$$

La fracción de aprendizaje r se define asumiendo que todas las conexiones de una neurona contribuyen de la misma forma en las correcciones que se realizan sobre la salida de una neurona:

$$\Delta y_j = -\frac{r}{n+1} \frac{\partial E}{\partial y_j}$$

donde n es el número de entradas de la neurona (sin contar la correspondiente al sesgo, de donde proviene el término $+1$).

De la expresión anterior se deriva el error inducido por una conexión particular y, de ahí, la ecualización que ha de realizarse. En el caso de las neuronas sigmoidales logísticas, la ecualización se consigue de la siguiente forma:

$$\eta_{ij} = \eta g_{ij} = \frac{16r_c}{n+1} \frac{1}{\epsilon_P[x_{ij}^2]}$$

donde $\epsilon_P[x_{ij}^2]$ representa el P -ésimo percentil de x_{ij}^2 , el cual sirve para aproximar la escala de x_{ij}^2 cuando P es lo suficientemente cercano a 100, y r_c es un “fracción de aprendizaje crítica”, un parámetro que se establece a priori y sirve de punto de referencia para todas las tasas locales de aprendizaje.

Para los sesgos de las neuronas, asociados a entradas fijas $x_{0j} = 1$, se introduce un parámetro adicional $G < 1$ para compensar por la discrepancia entre $\epsilon_P[x_{0j}^2] = 1$ y $\epsilon_P[x_{ij}^2] < 1$ para $i \neq 0$:

$$\eta_{0j} = \eta g_{0j} = \frac{16r_c}{n+1} \frac{G}{x_{0j}^2}$$

Como sucedía en algunas técnicas de inicialización de la escala de los pesos de la red, las tasas de aprendizaje resultantes son inversamente proporcionales a la raíz cuadrada de las escalas de las entradas y al *fan-in* de cada nodo.³⁴⁹

EEBP, además, propone sustituir la neuronas ocultas de forma que todas las distribuciones x_{ij}^2 tengan su moda cerca de cero. Esto lo consigue haciendo que, cuando la salida de la neurona y_j suele estar por encima de 0.5 (en el caso de la función logística), la neurona oculta se reemplaza por una neurona complementaria con pesos $-w_{ij}$ y salida

³⁴⁸ Jean-Pierre Martens y Nico Weymaere. An equalized error backpropagation algorithm for the on-line training of multilayer perceptrons. *IEEE Transactions on Neural Networks*, 13(3):532–541, May 2002. ISSN 1045-9227. DOI: 10.1109/TNN.2002.1000122

El factor 16 proviene de elevar al cuadrado el máximo de la derivada de la función logística ($1/4$), el cual desaparecería si utilizamos la tangente hiperbólica como función de activación, cuya derivada tiene un máximo de 1.

³⁴⁹ David C. Plaut y Geoffrey E. Hinton. Learning sets of filters using back-propagation. *Computer Speech and Language*, 2(1):35–61, 1987. ISSN 0885-2308. DOI: 10.1016/0885-2308(87)90026-X

$1 - y_j$, lo que además obliga a modificar los pesos de las neuronas que reciben la salida de la neurona sustituida (cambiando su signo en el caso de las unidades logísticas).

Como podemos ver, disponemos de múltiples alternativas a la hora de establecer tasas locales de aprendizaje. Desde el punto de vista teórico, se han derivado algunos resultados formales con respecto a las propiedades de convergencia de algunas de las estrategias descritas, como es el caso de Quickprop. Bajo determinadas condiciones, que se pueden imponer en el entrenamiento por lotes, se puede garantizar que el error disminuye en cada época del entrenamiento de la red.³⁵⁰

En la práctica, no obstante, es mucho más común utilizar el gradiente descendente estocástico (*online* o con minibatches). Entonces, no nos queda más remedio que ajustar de forma heurística las tasas locales de aprendizaje de cada parámetro de la red en función de la consistencia del gradiente del error para ese parámetro particular. En función del problema particular, distintas estrategias pueden funcionar mejor o peor. Se puede hacer que las tasas de aprendizaje decrezcan lineal o exponencialmente, usando una cota mínima de la que nunca se bajaría. Por ejemplo, podríamos hacer que, cada vez que el error de validación se estabiliza en una posible meseta, se reduzca la tasa de aprendizaje por un factor de 2 a 10.

Como hemos visto en esta sección la tasa de aprendizaje más adecuada puede variar por distintos factores, entre los que destacan las magnitudes de los gradientes o el *fan-in* de cada nodo. Ambos factores pueden ser muy diferentes en las distintas capas de una red multicapa, por lo que conviene tenerlos en cuenta a la hora de ajustar las tasas locales de aprendizaje.

Momentos

Además de establecer adecuadamente la escala de los pesos iniciales de la red y ajustar las tasas de aprendizaje, ya sea a nivel local o global, otra forma común de acelerar la convergencia del algoritmo de entrenamiento de redes neuronales consiste en el uso de momentos.

El uso de momentos añade un hiperparámetro más a la fórmula de actualización de los pesos de la red:

$$\Delta w(t) = -\eta \nabla E(w(t)) + \mu \Delta w(t-1)$$

El parámetro μ se denomina, incorrectamente, momento. En sentido físico, representa más bien una inercia: cuando se calcula el cambio al que debe someterse un peso de la red, se le añade una fracción del último cambio que sufrió ese parámetro. Intuitivamente, estamos añadiendo a la regla de actualización de los pesos información acerca de la segunda derivada del error. La segunda derivada del error, en forma de matriz

³⁵⁰ George D. Magoulas, Vassilis P. Plagianakos, y Michael N. Vrahatis. Globally convergent algorithms with local learning rates. *IEEE Transactions on Neural Networks*, 13(3):774-779, May 2002. ISSN 1045-9227. DOI: 10.1109/TNN.2002.1000148

Un observador sagaz se habrá dado cuenta de que esta estrategia ya la utilizamos cuando intentábamos estimar la tasa de aprendizaje óptima usando el *eigenvector* principal de la matriz Hessiana o empleábamos medias móviles en la regla delta-barra-delta.

Hessiana para el caso multidimensional, nos proporciona información no sólo acerca del gradiente del error, sino también acerca de cómo cambia ese gradiente. El uso de momentos utiliza la misma idea pero sin necesidad de calcular una matriz enorme de segundas derivadas (lo que, por otra parte, resulta del todo inviable cuando tenemos una red neuronal con millones de parámetros).

El uso de momentos ayuda a mantener las modificaciones de los pesos en la misma dirección. Aparte de evitar cambios bruscos en la dirección de búsqueda del mínimo, sorteando muchas oscilaciones, podemos verlo como una forma de evitar que el gradiente descendente se quede atascado en el primer mínimo local que encuentre, por poco pronunciado que sea. Desde el punto de vista físico, es como si dejamos caer una canica por una superficie inclinada. En su descenso llevará una velocidad que se ve afectada por la pendiente local del punto en el que nos encontramos y la fricción de la superficie sobre la que rueda. Con poca inercia, la canica se quedará en la primera irregularidad que encuentre en su recorrido. Con algo más de inercia, superará algunos baches (mínimos locales) hasta llegar a un mínimo que esperemos que resulte óptimo.

En la fórmula de actualización de los pesos, el gradiente del error ∇E modifica la velocidad como lo haría la fuerza gravitatoria que hace que una canica se deslice sobre una pendiente. El momento μ introduce fricción en el movimiento, tendiendo a reducir gradualmente su velocidad:

- Cuando $\mu = 1$, el movimiento se realiza sin fricción alguna, por lo que la única modificación sobre la velocidad del cambio en los pesos Δw proviene del gradiente del error, con lo que la inercia del movimiento hará que nos pasemos del mínimo en el que nos gustaría detenernos.
- Cuando $\mu = 0$ estamos en el caso típico del gradiente descendente, que corresponde a un movimiento con fricción absoluta que se queda atascado en el primer mínimo local que nos encontramos.
- Cuando μ toma un valor intermedio entre 0 y 1, la fricción del movimiento $(1 - \mu)$ nos permitirá conservar nuestra inercia en el movimiento pero, idealmente, no tanto como para pasarnos del mínimo que deseamos encontrar.

Por tanto, el parámetro μ asociado al momento tomará un valor positivo menor que 1. En sentido físico, debería haberse denominado inercia o fricción, pero no momento.

Definiendo de forma explícita la velocidad v del cambio en los parámetros de la red, Δw , el uso de momentos se puede representar como:

$$\begin{aligned} v(t+1) &= \mu v(t) - \eta \nabla E(w(t)) \\ \Delta w &= w(t+1) - w(t) = v(t+1) \end{aligned}$$

¿Qué pasaría si utilizamos un momento $\mu > 1$? ¿Y si nos equivocamos de signo y usamos $\mu < 0$?

Una modificación habitual del algoritmo basado en momentos, tal como los hemos definido, consiste en el uso de momentos de Nesterov.³⁵¹ En la versión tradicional de los momentos, se evalúa el gradiente del error en la posición actual y , además del movimiento asociado a ese gradiente, se da un salto en función del gradiente observado previamente. La versión mejorada, basada en el método de optimización de funciones convexas de Nesterov, primero da un salto en la dirección del gradiente previamente observado y luego evalúa el gradiente del error, en la posición a la que se llega tras el salto:

$$\begin{aligned} v(t+1) &= \mu v(t) - \eta \nabla E(w(t) + \mu v(t)) \\ \Delta w &= w(t+1) - w(t) = v(t+1) \end{aligned}$$

La corrección del gradiente se realiza tras el salto en la dirección que nos marca el momento. El momento de Nesterov, pues, se basa en corregir un error después de cometerlo, lo que parece tener más sentido desde el punto de vista intuitivo y contribuye a acelerar la convergencia del algoritmo basado en momentos.

Aunque hemos descrito los momentos μ como si fuesen parámetros globales del entrenamiento de la red, independientes de las tasas de aprendizaje, podemos incorporar momentos cuando usamos tasas locales de aprendizaje. Como resultado, se obtienen momentos locales μ_{ij} , específicos para cada parámetro de la red, que pueden combinarse con distintas estrategias de ajuste local de las tasas de aprendizaje. Por ejemplo, se puede diseñar una versión extendida de la regla delta-barra-delta en la que tanto las tasas de aprendizaje η_{ij} como los momentos μ_{ij} se ajustan dinámicamente.³⁵² Las tasas de aprendizaje adaptativas sólo tienen en cuenta los efectos alineados con los ejes (el gradiente del error con respecto a cada peso de la red), mientras que los momentos no imponen esa restricción. Al incorporar momentos a las tasas de aprendizaje adaptativas, se puede aprovechar tanto la concordancia de signo de los gradientes del error como la "velocidad" inercial de los cambios previos en los parámetros del aprendizaje.

Desde el punto de vista teórico, como sucede con otras estrategias de tipo heurístico, también se han analizado las condiciones de convergencia del uso de momentos en el entrenamiento de una red neuronal.³⁵³

Incluso se han llegado a proponer diferentes métodos "óptimos" para ajustar conjuntamente las tasas de aprendizaje η y los momentos μ en algunas situaciones particulares.^{354,355} Estos métodos, además de los cálculos típicos del gradiente descendente con *backpropagation*, calculan las derivadas del error con respecto a η y μ , por lo que pueden requerir hasta el triple de recursos computacionales que el algoritmo convencional. A cambio, prometen acelerar la convergencia del algoritmo en las situaciones en que son aplicables.

³⁵¹ Ilya Sutskever, James Martens, George Dahl, y Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. En Sanjoy Dasgupta y David McAllester, editores, *ICML'13 Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/sutskever13.pdf>

³⁵² A.A. Minai y R.D. Williams. Backpropagation heuristics: A study of the extended delta-bar-delta algorithm. En *1990 IJCNN International Joint Conference on Neural Networks*, pages 595–600 vol.1, June 1990. DOI: 10.1109/IJCNN.1990.137634

³⁵³ Jian Wang, Jie Yang, y Wei Wu. Convergence of cyclic and almost-cyclic learning with momentum for feedforward neural networks. *IEEE Transactions on Neural Networks*, 22(8):1297–1306, Aug 2011. ISSN 1045-9227. DOI: 10.1109/TNN.2011.2159992

³⁵⁴ Xiao-Hu Yu, Guo-An Chen, y Shi-Xin Cheng. Dynamic learning rate optimization of the backpropagation algorithm. *IEEE Transactions on Neural Networks*, 6(3):669–677, May 1995. ISSN 1045-9227. DOI: 10.1109/72.377972

³⁵⁵ Xiao-Hu Yu y Guo-An Chen. Efficient backpropagation learning using optimal learning rate and momentum. *Neural Networks*, 10(3):517 – 527, 1997. ISSN 0893-6080. DOI: 10.1016/S0893-6080(96)00102-5

Otras formas de actualizar los parámetros de la red

El uso de momentos, para lo que añadimos un nuevo término a la función de actualización de los parámetros de la red, sirve para abrir la caja de Pandora de las fórmulas que podemos utilizar para ir actualizando los parámetros de una red durante su entrenamiento.

Si denotamos por δ_{ij} al valor de la derivada parcial de la función de error con respecto al parámetro w_{ij} de la red, $\delta_{ij} = \partial E / \partial w_{ij}$, el método de optimización del gradiente descendente se puede expresar, en función de una tasa de aprendizaje η , como

$$\Delta w_{ij}(t) = -\eta \delta_{ij}(t)$$

Si añadimos un momento μ , la fórmula de actualización de los pesos pasa a ser

$$\Delta w_{ij}(t) = -\eta \delta_{ij}(t) + \mu \Delta w(t-1)$$

Además del gradiente del error y del momento, podemos añadir un tercer término proporcional, en analogía con los controladores PID que se utilizan en sistemas de control.³⁵⁶

$$\Delta w_{ij}(t) = -\eta \delta_{ij}(t) + \mu \Delta w(t-1) + \gamma E(t)$$

Aplicable tanto en aprendizaje por lotes como cuando se utiliza el gradiente descendente estocástico (por minilotes o incremental/*online*), suele converger más rápidamente que el algoritmo convencional, escapar de mínimos locales con mayor facilidad y ser más robusto con respecto a la selección de los parámetros iniciales de la red. A costa, claro está, de introducir un hiperparámetro más, γ , que tendremos que ajustar.

No se trata, ni mucho menos, de la única variación que se ha propuesto sobre el tema. Por poner un ejemplo curioso, el *backpropagation* emocional³⁵⁷ (sic) añade neuronas emocionales que proporcionan nuevas entradas al resto de las neuronas de la red. Esas conexiones de las neuronas emocionales se ajustan con dos parámetros adicionales: un coeficiente de ansiedad α y un coeficiente de confianza κ . Las referencias psicológicas resultan evidentes. Cuando aprendemos una nueva tarea, nuestra ansiedad es alta y nuestra confianza, baja. Con la práctica, nuestra ansiedad disminuye y nuestra confianza aumenta. La actualización de los pesos de la red se realiza combinando los cambios convencionales en los pesos, Δw_c , con los cambios emocionales de la red, Δw_m , en los que intervienen sus niveles de ansiedad y confianza.

En la tabla se muestran otras reglas de aprendizaje que se pueden derivar de la regla delta generalizada, que en algunos ámbitos se conoce con el nombre de algoritmo LMS [*Least Mean Squares*].³⁵⁸

- En una actualización con pérdidas [*leaky*], un factor de pérdida $\gamma < 1$ hace que la red olvide lo aprendido previamente. Si $\gamma = 0$, tenemos

³⁵⁶ Y.H. Zweiiri, J.F. Whidborne, y L.D. Seneviratne. A three-term backpropagation algorithm. *Neurocomputing*, 50: 305 – 318, 2003. ISSN 0925-2312. DOI: 10.1016/S0925-2312(02)00569-6

³⁵⁷ Adnan Khashman. A modified backpropagation learning algorithm with added emotional coefficients. *IEEE Transactions on Neural Networks*, 19(11): 1896–1909, Nov 2008. ISSN 1045-9227. DOI: 10.1109/TNN.2008.2002913

³⁵⁸ Bernard Widrow y Marcian E. Hoff. Adaptive switching circuits. En *1960 IRE WESCON Convention Record, Part 4*, pages 96–104, New York, August 1960. Institute of Radio Engineers, Institute of Radio Engineers. URL <http://www-isl.stanford.edu/~widrow/papers/c1960adaptiveswitching.pdf>

Regla	Actualización de los pesos
Gradiente descendente (LMS)	$\Delta w_{ij}(t+1) = -\eta x_i \delta_j$ $\eta > 0$
Con pérdidas (<i>leaky LMS</i>)	$\Delta w_{ij}(t+1) = -\eta x_i \delta_j - \gamma w_{ij}$ $\eta > 0, 0 \leq \gamma < 1$
Momentos	$\Delta w_{ij}(t+1) = -\eta x_i \delta_j + \mu \Delta w_{ij}(t)$ $\eta > 0, \mu \geq 0$
MLMS (normalizado)	$\Delta w_{ij}(t+1) = -\eta \frac{x_i + \delta_j}{\alpha + \ x_i\ ^2}$ $\eta > 0, \alpha \geq 0$
RWLS (ARMA)	$\Delta w_{ij}(t+1) = -\eta(t) x_i \delta_j P(t)$ $\eta(t) = \frac{1}{\lambda + x^T(t) P(t) x(t)}$ $P(t+1) = \frac{[I - \eta(t) P(t) x(t) x^T(t)] P(t)}{\lambda}$ $\eta > 0, 0 < \lambda < 1$

Tabla 6: Distintas reglas de aprendizaje.

la regla habitual que hemos utilizado con el gradiente descendente. Cuanto mayor sea γ , más rápidamente se olvidará de lo aprendido, de ahí que, en caso de emplearse, se suelan utilizar valores muy pequeños para este parámetro (p.ej. 10^{-8}).

- La versión normalizada [*NLMS: Normalized LMS*] se basa en el principio de mínima perturbación del ADALINE: cuando se adaptan los pesos para responder adecuadamente a nuevos patrones en la entrada, las respuestas de la red a patrones previos se modifican lo mínimo posible (en media). Es una forma de ajuste automático de la tasa de aprendizaje, una autonormalización para que el ajuste no dependa de la magnitud ni del número de entradas. Si las entradas ya llegan normalizadas, equivale a la regla estándar cuando $\alpha = 0$. En el denominador se suele introducir esa constante positiva α para asegurarnos de que las actualizaciones estén siempre acotadas, evitando dividir por cero en ausencia de entradas. Observe, además, que si las entradas son bipolares, $\{+1, -1\}$, el término $\|x_i\|^2$ no es más que el número de entradas de la neurona, un ajuste típico que ya hemos visto que se utiliza a la hora de ajustar las escalas iniciales de los pesos y las tasas de aprendizaje.
- La regla más compleja, RWLS [*Recursive Weighted Least Squares*], es una forma más de establecer tasas de aprendizaje adaptativas. La regla se basa en el uso de modelos autorregresivos de medias móviles [*ARMA: AutoRegressive Moving Average*], también llamados modelos de Box-Jenkins, una técnica estadística que se emplea en el análisis de series temporales y en el diseño de filtros adaptativos. El modelo involucra el uso de una matriz simétrica de ponderación P que, para el caso que nos ocupa, puede ser la matriz identidad, p.ej. $I_{3,3}$.³⁵⁹

³⁵⁹ Fredric M. Ham y Ivica Kostanic. *Principles of Neurocomputing for Science and Engineering*. McGraw-Hill, 2000. ISBN 007025966

Los métodos NLMS y RWLS, aunque puedan parecer enrevesados a simple vista, están diseñados para reducir el número necesario de iteraciones del algoritmo para alcanzar la convergencia, a menudo de forma significativa, lo que puede acelerar el proceso de entrenamiento de una red neuronal artificial. No son los únicos, ya que existen múltiples técnicas de cálculo numérico que pueden emplearse en la resolución de problemas de optimización de este tipo.³⁶⁰ Las técnicas numéricas de optimización suelen ser aplicables al entrenamiento de redes neuronales cuando utilizamos aprendizaje por lotes, si bien también existen algunas que se pueden emplear con el gradiente descendente estocástico³⁶¹ o admiten tasas locales de aprendizaje.³⁶²

Otra alternativa más radical consiste en utilizar un algoritmo *greedy* que vaya entrenando los parámetros de la red capa por capa: OLL [*Optimization Layer by Layer*].³⁶³ En vez de emplear un algoritmo iterativo de optimización, se pueden “linealizar” los elementos no lineales de una red multicapa y emplear esa linealización para ajustar matemáticamente los parámetros de una capa de la red multicapa. De esta forma, el entrenamiento de una red multicapa se convierte en el entrenamiento de una serie de modelos que sólo tienen una capa de parámetros ajustables. Básicamente, la misma estrategia que proponían las primeras técnicas de *deep learning*, sin llegar al extremo de simplificar nuestro modelo hasta que sólo tenga una capa de parámetros ajustables, como sucede con las máquinas de vectores de soporte [*SVM: Support Vector Machines*] o las máquinas de aprendizaje extremo [*ELM: Extreme Learning Machines*].

En la práctica, una selección razonable de algoritmo de entrenamiento suele consistir en utilizar el gradiente descendente estocástico (ya sea con minibatches u *online*) con momentos y tasas de aprendizaje adaptativas que vayan descendiendo, lineal o exponencialmente, conforme progresa el aprendizaje de los parámetros de la red.

Normalización por lotes

Para cerrar nuestro repaso general a las heurísticas que se pueden utilizar en el entrenamiento de redes neuronales artificiales, comentaremos una técnica muy utilizada en *deep learning*: la normalización por lotes [*batch normalization*].

La normalización por lotes, propuesta en 2015 por dos investigadores de Google, Sergey Ioffe y Christian Szegedy, proporciona un mecanismo de reparametrización adaptativa para facilitar el entrenamiento de redes profundas.³⁶⁴ Se encuentra a medio camino entre la normalización de los datos de entrada, la selección de la escala adecuada para la inicialización de los pesos de la red y las estrategias de ajuste automático de las tasas locales de aprendizaje. Su objetivo, como el de todas esas técnicas, es evitar que el entrenamiento de la red se ralentice debido a las diferencias

³⁶⁰ Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific Publishers, 2nd edition, 1999. ISBN 1886529000

³⁶¹ Rainer Gemulla, Erik Nijkamp, Peter J. Haas, y Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. En *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, 2011. ISBN 978-1-4503-0813-7. DOI: 10.1145/2020408.2020426

³⁶² Gábor Takács, István Pilászy, Bottyán Németh, y Domonkos Tikk. Matrix Factorization and Neighbor Based Algorithms for the Netflix Prize Problem. En *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pages 267–274, 2008. ISBN 978-1-60558-093-7. DOI: 10.1145/1454008.1454049

³⁶³ S. Ergezinger y E. Thomsen. An accelerated learning algorithm for multilayer perceptrons: optimization layer by layer. *IEEE Transactions on Neural Networks*, 6(1):31–42, Jan 1995. ISSN 1045-9227. DOI: 10.1109/72.363452

³⁶⁴ Sergey Ioffe y Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. En Francis Bach y David Blei, editores, *ICML'2015 Proceedings of the 32nd International Conference on Machine Learning*, volume 37 de *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR. URL <http://proceedings.mlr.press/v37/ioffe15.html>

que surgen entre las diferentes capas de una red con múltiples capas ocultas, algo a lo que Ioffe y Szegedy denominan “variación covariable interna” [*internal covariate shift*]. Sin el uso de mecanismos de este tipo, el gradiente descendente usando *backpropagation* es difícil que consiga entrenar con éxito redes profundas.

Dado que el entrenamiento de una red profunda se complica porque la distribución de las entradas de una capa cambia durante el entrenamiento conforme se ajustan los parámetros de las capas anteriores, la normalización por lotes consiste en normalizar las entradas de cada capa. En lugar de normalizar sólo las entradas de la red, la normalización por lotes se aplica en toda la red. Se puede interpretar como un mecanismo de preprocesamiento de los datos de entrada de cada capa de la red.

Para cada minilote, básicamente, estimamos la media μ y la varianza σ^2 de cada uno de los parámetros que nos interesen. A continuación, normalizamos usando *z-scores*. Por ejemplo, dados los niveles de activación h de una neurona oculta para un minilote de m ejemplos, su versión normalizada por lotes, h_{bn} , se calcula como

$$\begin{aligned}\mu &= \frac{1}{m} \sum_{i=1}^m h_i \\ \sigma^2 &= \frac{1}{m} \sum_{i=1}^m (h_i - \mu)^2 \\ h_{bn} &= \frac{h - \mu}{\sqrt{\sigma^2 + \epsilon}}\end{aligned}$$

donde ϵ es un valor muy pequeño (p.ej. $\epsilon = 10^{-8}$), utilizado simplemente para evitar divisiones por cero.

Una vez entrenada la red, los valores de μ y δ pueden reemplazarse por medidas recogidas durante el proceso de entrenamiento para utilizar la red con nuevos datos de entrada.

Esta normalización, tal cual, podría afectar al comportamiento de la red. Por ejemplo, si normalizamos así las entradas de una neurona sigmoidal, puede que estemos obligando a que opere en su régimen lineal. Por este motivo, a la normalización anterior se le suele añadir una transformación lineal $\gamma h_{bn} + \beta$ que garantice que se puede representar la función identidad. El resultado de esta transformación es $BN_{\gamma, \beta}(h) = \gamma h_{bn} + \beta$, donde γ y β se aprenden como cualquier otro parámetro de la red, al ser la normalización una sencilla operación diferenciable. Es decir, la normalización por lotes preprocesa los datos con los que se trabaja en toda la red pero, en lugar de diseñar mecanismos específicos para ello, este preprocesamiento se integra en el propio algoritmo de entrenamiento de la red, basado en ajustar sus parámetros de acuerdo al gradiente del error.

La normalización por lotes, por tanto, estandariza la media y la varianza de cada unidad de la red, consiguiendo un efecto similar al que

De hecho, si fijamos $\gamma = \sigma$ y $\beta = \mu$, obtendríamos los niveles de activación originales, si eso fuese lo que nos conviene hacer.

lograríamos si dinámicamente seleccionamos una tasa local de aprendizaje adecuada, algo difícil de conseguir en una red profunda porque los efectos de un cambio en los parámetros de una capa dependen de los parámetros de todas las capas que la preceden. La estandarización estabiliza el comportamiento del algoritmo de entrenamiento de la red, sin impedir el aprendizaje ni eliminar artificialmente el comportamiento no lineal de la red.

De hecho, una forma de intentar conseguir el mismo efecto, que inspiró la normalización por lotes, consistía en eliminar las relaciones lineales entre distintas neuronas dentro de una misma capa, como en PRONG [*PRProjected Natural Gradient descent algorithm*].³⁶⁵ Por desgracia, eliminar las interacciones lineales dentro de una capa es mucho más difícil que estandarizar la media y la varianza de cada neurona individual, motivo por el que, en la práctica, se utiliza la normalización por lotes y no otros mecanismos de reparametrización más complejos.

Para implementar la normalización por lotes, podemos intercalar, entre las capas de la red, nuevas capas de normalización que se encarguen de reemplazar los niveles de activación de las unidades ocultas h por $BN_{\gamma,\beta}(h) = \gamma h_{bn} + \beta$. Los parámetros γ y β se aprenden igual que los pesos de las demás capas, con el objetivo de permitir que las variables normalizadas puedan tener cualquier media y desviación estándar. De esta forma, $BN_{\gamma,\beta}(h)$ representa la misma familia de funciones que los valores h originales. En su versión original, la media de h dependía de las interacciones complejas entre muchísimos parámetros de la red. En su versión normalizada por lotes, depende simplemente de β . Su reparametrización hace que el entrenamiento de la red funcione mucho mejor usando el gradiente descendente.

¿Dónde se aplica la normalización por lotes? Podríamos aplicarla sobre los niveles de activación de las neuronas ocultas de la red, que sirven de entrada x a las capas siguientes. Sin embargo, se recomienda aplicarla a la entrada neta de las neuronas $z = wx + b$, lo que además nos permite prescindir del sesgo (el sesgo b resulta redundante con el parámetro β de la normalización por lotes).

La normalización por lotes, en definitiva, es muy sencilla de implementar y soluciona muchos de los problemas que nos obligaban a diseñar estrategias sofisticadas de inicialización de los parámetros de la red y de ajuste local de sus tasas de aprendizaje. En problemas reales, como en el reconocimiento de objetos en imágenes usando redes convolutivas, ha demostrado ser mucho más eficiente que otras alternativas. Hace a la red más robusta frente a problemas en la inicialización de sus parámetros y, además, suele tener un efecto regularizador sobre la red una vez entrenada; esto es, mejora su capacidad de generalización.

³⁶⁵ Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, y Koray Kavukcuoglu. Natural neural networks. En Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, y Roman Garnett, editores, *NIPS'2015 Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2071–2079, 2015. URL <http://papers.nips.cc/paper/5953-natural-neural-networks>

Hiperparámetros

Hemos visto que se puede ajustar la topología de una red neuronal modificando su número de capas o el número de neuronas por capa, se dispone de múltiples alternativas de funciones de activación para modelar el funcionamiento de neuronas individuales, existen distintos modos de entrenamiento de una red, se explota de diferentes formas la información recogida en el conjunto de datos de entrenamiento, se preparan los parámetros iniciales de la red de un sinfín de maneras y se pergeñan múltiples estrategias para el ajuste de los parámetros de la red, por no hablar ya de distintas reglas de aprendizaje que se pueden emplear. El número de grados de libertad de los que uno dispone para resolver un problema con redes neuronales artificiales es muy elevado y el número resultante de posibles combinaciones es prácticamente infinito. Si a todo esto le añadimos el hecho de que una configuración adecuada puede resolver el problema pero una incorrecta puede dar resultados manifiestamente mejorables, ¿qué hacemos al respecto?

Es posible que, en su cabeza, le esté dando vueltas al problema después de haber leído acerca de múltiples sugerencias y recomendaciones de tipo heurístico en las secciones anteriores. Por desgracia, sólo son eso: criterios de tipo heurístico. El diseño, entrenamiento y depuración de una red neuronal, como sucede con la depuración de un programa de ordenador, tiene una parte de ciencia pero mucho de arte. Para obtener buenos resultados, hay que dominar el arte de elegir una buena topología para la red neuronal y establecer de forma adecuada los múltiples hiperparámetros que gobiernan su entrenamiento.

Las redes neuronales artificiales, como modelos de aprendizaje automático, incluyen un gran número de parámetros que se ajustan durante su entrenamiento. El propio proceso de entrenamiento dispone de muchos parámetros que también hemos de ajustar: los hiperparámetros de la red. Desafortunadamente para nosotros, esos hiperparámetros muchas veces interactúan de formas complejas, lo que convierte el entrenamiento de una red en un arduo proceso para el que resulta extremadamente difícil establecer recomendaciones generales. Aun así, veamos qué podemos hacer al respecto, manualmente y con ayuda de herramientas que automaticen parcialmente el proceso.

Ajuste manual

Al establecer manualmente los hiperparámetros de una red neuronal, generalmente, lo que estamos intentando conseguir es controlar la capacidad de la red para adaptarla al conjunto de datos del que disponemos. A grandes rasgos, los hiperparámetros que podemos controlar los podemos agrupar en tres grandes grupos:

Es habitual que, para un mismo problema, distintas herramientas de *deep learning*, que aparentemente implementan el mismo tipo de técnicas, proporcionen resultados muy diferentes. Esas diferencias estriban, habitualmente, en cómo se hayan establecido los diferentes hiperparámetros con los que podemos ajustar el entrenamiento de una red neuronal.

- Hiperparámetros universales, comunes prácticamente a cualquier tipo de red neuronal artificial, entre los que destacan los relacionados con la topología de la red (número de capas de la red, número de neuronas por capa y patrones de interconexión entre las distintas capas), los asociados al modo de entrenamiento (p.ej. tamaño de los minibatches) y los parámetros asociados a la técnica de optimización que utilicemos para ajustar los parámetros de la red (p.ej. tasas de aprendizaje y momentos).
- Hiperparámetros específicos para determinadas arquitecturas de redes neuronales, como pueden ser el tamaño de los *kernels* o las estrategias de *pooling* en las redes convolutivas.
- Hiperparámetros asociados al algoritmo de entrenamiento concreto que utilicemos y a múltiples estrategias de regularización con las que podemos intentar mejorar la capacidad de generalización de la red (normalización por lotes, *weight decay*, *dropout*...).

Aunque algunos de esos hiperparámetros no los hemos estudiado aún, muchos de ellos se pueden establecer razonando acerca de si aumentan o disminuyen la capacidad de la red. En función de los resultados que obtengamos sobre los conjuntos de entrenamiento, validación y prueba, podemos intentar deducir en qué sentido habría que modificar cada hiperparámetro.

Ahora bien, el entrenamiento de una red neuronal grande, diseñada para resolver un problema complejo, puede requerir mucho tiempo. Por este motivo, las primeras pruebas deberíamos hacerlas con una versión simplificada del problema, tal vez usando sólo un subconjunto de los datos disponibles, de forma que podamos realizar experimentos más rápidamente. La estrategia general consiste en atacar el problema (o su versión simplificada) hasta conseguir una versión inicial de la red que cumpla mínimamente su función. Al menos, que obtenga mejores resultados que los que podríamos obtener al azar. Una vez ahí, podemos comenzar un proceso iterativo de prueba y error en el que, monitorizando los resultados obtenidos, nos permita ir realizando mejoras incrementales sobre el sistema.

En nuestra implementación, podemos utilizar una infraestructura que nos facilite la realización continua de experimentos con múltiples conjuntos de hiperparámetros, tal vez en un cluster de ordenadores o en una máquina equipada con una GPU que permita acelerar los cálculos. Dado que el proceso puede ser costoso, conviene que dispongamos de los mecanismos para almacenar en un fichero tanto los resultados que vayamos obteniendo como los parámetros en sí de la red, para poder reutilizarlos cuando estimemos oportuno sin tener que repetir el proceso completo de entrenamiento. Aunque podríamos realizar una validación cruzada,

Curiosamente, el tamaño de las redes ha aumentado conforme se iba disponiendo de mayores conjuntos de datos y más capacidad de cálculo, pero el tiempo de entrenamiento utilizado en sistemas reales parece mantenerse relativamente constante: un par de semanas, desde las redes más grandes que se usaban a principios de los años 90 para el control de calidad de superficies pintadas (con una sola capa oculta, menos de 1000 neuronas y sólo 45k conexiones sinápticas, prácticamente enanas de acuerdo a los estándares actuales) hasta AlexNet, utilizada en 2012 para ganar la competición de clasificación de imágenes de ImageNet (una red convolutiva con múltiples capas ocultas, 650,000 neuronas y 60 millones de parámetros).

resulta más económico utilizar un único conjunto de entrenamiento y un único conjunto de validación. El conjunto de validación, independiente del conjunto de entrenamiento, es el que usaremos de guía para ajustar los hiperparámetros de la red.

Para poder revisar y monitorizar el comportamiento del proceso de entrenamiento, también nos vendrá bien disponer de información relativa a cómo va variando la red en cada época de su entrenamiento (error sobre el conjunto de entrenamiento, magnitud de los gradientes por capas, escalas de los pesos y de las tasas de aprendizaje...), que podemos analizar mejor si la representamos de forma gráfica.

Una vez dispongamos de la infraestructura necesaria para ir realizando experimentos, tendremos que buscar la combinación de hiperparámetros más adecuada para nuestro problema particular. Podemos empezar con una estimación a trazo grueso de los valores de los distintos hiperparámetros para establecer su escala más adecuada y, a continuación, intentar afinar cada uno de ellos. En ocasiones, ni siquiera necesitaremos completar el entrenamiento completo de la red, ya que cualquier desviación de su comportamiento deseable nos permitirá descartar determinadas combinaciones (p.ej. cuando en una o varias épocas la red es incapaz de mejorar o el algoritmo de entrenamiento resulte inestable, disparándose el error, los gradientes o los pesos). En las etapas iniciales de nuestro ajuste podemos realizar experimentos usando un número reducido de épocas de entrenamiento e ir incrementando éste conforme vayamos afinando mejor los distintos hiperparámetros.

En muchas ocasiones, la tasa de aprendizaje es el parámetro más importante en el entrenamiento de una red neuronal, lo que justifica que sea el hiperparámetro al que más espacio le hemos dedicado en las páginas anteriores. Ya sea una tasa de aprendizaje global o un valor de referencia que se utilice para establecer la escala de las tasas locales de aprendizaje, resulta fundamental establecer adecuadamente este parámetro para que la red aprenda correctamente.

La tasa de aprendizaje controla la capacidad efectiva de la red. Si es demasiado elevada, la red puede que ni aprenda (el error puede aumentar en lugar de disminuir usando el gradiente descendente). Si es demasiado baja, la red puede que aprenda, pero demasiado despacio, suponiendo que no se quede atascada con una tasa de error demasiado elevada.

Para establecer la tasa de aprendizaje adecuada, primero se prueban tasas de distintas escalas (p.ej. $10^{[-6,-1]}$), multiplicando y dividiendo por 10 su valor para determinar el umbral a partir del cual la función de error sobre el conjunto de entrenamiento decrece en vez de oscilar o aumentar. En función de los resultados obtenidos, iremos estrechando el rango de la tasa de aprendizaje.

Como ya vimos anteriormente, también puede resultar útil comenzar el entrenamiento utilizando una tasa de aprendizaje grande que permita

El fenómeno de quedarse atascada permanentemente en una tasa de error subóptima, algo que nunca ocurriría con una función de error convexa, se observa a menudo en la práctica.

que los pesos de la red cambien rápido y, luego, conforme avanza el entrenamiento de la red, utilizar algún esquema de reducción de la tasa de aprendizaje para ir afinando los valores finales de los pesos en la red entrenada.

Ajuste automático

Una red neuronal es un modelo con múltiples parámetros, cuyos valores se ajustan mediante un algoritmo de entrenamiento con la ayuda de un conjunto de datos de entrenamiento. Los hiperparámetros son todos aquellos parámetros, de la red o del propio algoritmo de entrenamiento, que el algoritmo de entrenamiento no ajusta automáticamente. De la misma forma que el algoritmo de entrenamiento ajusta automáticamente los parámetros del modelo de red neuronal que estemos utilizando, podemos diseñar algoritmos que automaticen la selección de todos aquellos parámetros que intervienen en el diseño y entrenamiento de una red neuronal pero de los que no se hace cargo el algoritmo de entrenamiento de la red. Esos algoritmos se denominan técnicas de búsqueda de hiperparámetros [*hyperparameter search*] o, de forma más concisa, técnicas de autoajuste [*autotuning*].

La búsqueda manual del conjunto óptimo de hiperparámetros es una tarea ardua y laboriosa, por lo que no debería sorprendernos que se hayan propuesto múltiples técnicas para automatizar, aunque sea sólo parcialmente, este proceso. En el proceso de entrenamiento de la red, el error sobre el conjunto de entrenamiento nos sirve de guía para ir ajustando los parámetros de la red. En el proceso de optimización de los hiperparámetros, usualmente no disponemos de una señal que nos indique directamente en qué sentido modificar los diferentes hiperparámetros, por lo que se suele recurrir a técnicas de búsqueda.

Desde el punto de vista del aprendizaje automático, se trata también de un problema de aprendizaje, salvo que ahora no estamos ajustando los parámetros de un modelo, sino los hiperparámetros que gobiernan el aprendizaje del modelo. De ahí que haya quien lo considere un problema de meta-aprendizaje, una forma de aprender a aprender. Hay quien parece pretender intentar ocultarlo bajo un halo de misterio, casi mágico. Sin embargo, más allá de la terminología que se use para describirlo, sus secretos no son inescrutables.

A diferencia del entrenamiento de los parámetros de una red, para el que disponíamos de un criterio general para ajustarlos (movernos en la dirección del gradiente descendente del error), la configuración óptima de los hiperparámetros depende del conjunto de datos particular. En este caso, resolvemos también un problema de optimización (con respecto a un conjunto de validación independiente), aunque lo hacemos mediante un algoritmo de búsqueda. En pseudocódigo, el proceso completo lo

Dado que el algoritmo de aprendizaje utilizado para entrenar un modelo no ajusta sus hiperparámetros, hay quien hace referencia a ellos como parámetros “*incordio*” [*nuisance parameters*]. No son específicos de las redes neuronales, sino que también existen en otras técnicas de aprendizaje automático.

El prefijo griego *meta-* hace referencia a un aumento en nuestro nivel de abstracción, a la formación nuevas abstracciones a partir de otros conceptos (metadatos, metamodelos, metamodelos...). Es uno de los recursos retóricos favoritos de muchos académicos, para darle más empaque a la resolución de un problema, incluso cuando ésta se base en el uso de técnicas más sencillas que las utilizadas para resolver los problemas sin el *meta-*, como sucede en el caso de las redes neuronales.

podríamos describir de la siguiente forma:

```
function (hp, model) = autotune (training, validation, settings)

(bestResults, bestHP, bestModel) = (null, null, null);

for each hp in settings:
    model = train (training, hp);
    results = evaluate (model, validation);
    if results better than bestResults:
        (bestResults, bestHP, bestModel) = (results, hp, model);

return (bestHP, bestModel);
```

donde `training` es un conjunto de datos de entrenamiento, `validation` es un conjunto de validación y `settings` es una lista con distintas configuraciones de los hiperparámetros `hp` del algoritmo de entrenamiento. Un modelo se entrena llamando a la función `train` y se evalúa con `evaluate`.

Tras probar diferentes configuraciones de los hiperparámetros, se devuelve la mejor configuración encontrada para los mismos junto con el modelo entrenado usando esa configuración. El proceso es sencillo desde el punto de vista lógico, aunque costoso computacionalmente, ya que cada iteración involucra entrenar una red neuronal completa. Se trata de un problema de optimización dentro de otro problema de optimización. La optimización de los hiperparámetros es mucho más difícil que el entrenamiento de la red porque, desde el punto de vista del proceso de optimización exterior, el entrenamiento de la red funciona como una caja negra. Para el ajuste de los hiperparámetros no existe una fórmula analítica que nos indique de forma explícita en qué dirección deberíamos proseguir nuestra búsqueda, como sí sucede con la función de error que guía el entrenamiento de una red.

Las técnicas existentes de ajuste automático de los hiperparámetros se pueden clasificar, grosso modo, en búsqueda sistemática [*grid search*], búsqueda aleatoria [*random search*] y búsqueda inteligente [*smart search*]. Todas esas técnicas buscan en el espacio de posibles configuraciones para encontrar buenas combinaciones de los múltiples hiperparámetros del entrenamiento de una red. Sus diferencias se hallan en la estrategia de búsqueda utilizada por cada una de ellas.

- *Búsqueda sistemática [grid search]*

La búsqueda sistemática realiza una búsqueda exhaustiva para conjuntos de valores de los hiperparámetros preestablecidos de antemano.

Usualmente, se selecciona un conjunto de n valores espaciados de forma uniforme en el rango de valores permitidos para cada hiperpará-

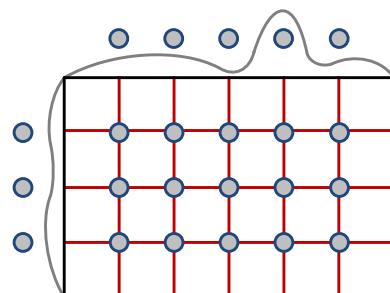


Figura 136: Ajuste de los hiperparámetros utilizando una búsqueda sistemática [*grid search*].

metros, ya sea utilizando una escala lineal (1, 2, 3, 4...10) o una escala logarítmica (0.001, 0.01, 0.1...).

Los valores seleccionados para los distintos hiperparámetros definen una rejilla [*grid*] en el espacio de posibles configuraciones y el proceso de búsqueda realiza una exploración sistemática en toda la rejilla, de ahí lo de *grid search*.

Si tenemos k hiperparámetros diferentes y, para cada uno de ellos, seleccionamos n valores diferentes, tendremos que evaluar n^k configuraciones diferentes. La búsqueda sistemática es el método de búsqueda de hiperparámetros más costoso computacionalmente, si bien es extremadamente simple y su paralelización es trivial.

Además de resultar muy costoso computacionalmente, requiere que preselecciónemos de forma adecuada qué valores son más prometedores para cada uno de los hiperparámetros del algoritmo. Si no estamos muy seguros de cuáles son, podemos realizar un proceso manual de búsqueda [*manual grid search*]. El proceso consiste en realizar una búsqueda sistemática en un *grid* reducido, que no sea demasiado grande en cuanto al número de configuraciones diferentes que se evalúan, y, en función de los resultados obtenidos, repetir iterativamente la búsqueda sobre diferentes *grids* hasta quedar satisfechos con los resultados.

■ Búsqueda aleatoria [*random search*]

La búsqueda aleatoria consiste en elegir aleatoriamente distintas configuraciones para los valores de los hiperparámetros. Aunque a priori pueda parecer sorprendente, la búsqueda aleatoria es mucho más eficiente en la optimización de hiperparámetros que la búsqueda sistemática.³⁶⁶ Además, es muy sencilla de implementar.

La búsqueda aleatoria es más eficiente tanto desde el punto de vista práctico como desde el punto de vista teórico. ¿Por qué? Porque, para un mismo número de configuraciones evaluadas, la búsqueda aleatoria explora un espacio de búsqueda mayor. A menudo, sólo algunos de los hiperparámetros influyen significativamente en los resultados finales obtenidos, aunque cuáles de ellos son realmente importantes pueden cambiar de un problema a otro. La búsqueda sistemática les dedica a todos los hiperparámetros el mismo esfuerzo, explorando exhaustivamente todas las configuraciones posibles. En cambio, la búsqueda aleatoria tarda menos que la búsqueda sistemática en encontrar alguna buena combinación de los hiperparámetros que realmente importan.

La búsqueda aleatoria lo único que hace es sustituir el conjunto fijo de muestras de la búsqueda en un *grid* por una muestra aleatoria. Antes del artículo de Bergstra y Bengio nadie la tomaba demasiado en serio, pero ellos supieron justificar por qué resulta recomendable.

³⁶⁶ James Bergstra y Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13(1):281–305, February 2012. ISSN 1532-4435. URL <http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>

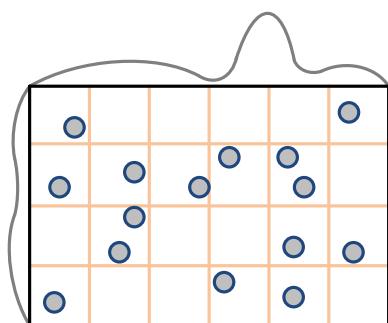


Figura 137: Ajuste de los hiperparámetros utilizando una búsqueda aleatoria [*random search*].

Para cualquier distribución con un máximo finito, el máximo de una muestra aleatoria de 60 observaciones está, como mucho, al 5 % del máximo auténtico con una probabilidad del 95 %. ¿De dónde sale esto? Imaginemos un intervalo en el entorno del máximo que cubra el 5 % del espacio muestral. Una muestra aleatoria tendrá una probabilidad del 5 % de caer dentro de dicho intervalo y del 95 % de no hacerlo. Si elegimos n muestras, la probabilidad de que ninguna de ellas caiga en ese intervalo será $(1 - 0.05)^n$. La probabilidad de que alguna acierte será de $1 - 0.95^n$, que es superior a 0.95 si $n \geq 60$.

En resumen, si el 5 % de las posibles configuraciones de hiperparámetros nos permiten obtener una solución cercana a la óptima, una simple búsqueda aleatoria con sólo 60 intentos nos permitirá encontrar alguna de ellas con una probabilidad elevada (superior al 95 %). Para ello, obviamente, necesitamos que el porcentaje de posibles configuraciones que proporcionan buenas soluciones sea relativamente elevado. Esto suele suceder cuando hay muchos hiperparámetros que no resultan realmente determinantes y lo podemos conseguir si utilizamos alguna estrategia de entrenamiento de la red que sea robusta frente a diferencias en sus condiciones iniciales.

La búsqueda aleatoria nos evita, de una forma sencilla, la necesidad de tener que elegir manualmente valores que sean realmente prometedores para los distintos hiperparámetros. Si nuestra selección es excepcionalmente buena, la búsqueda sistemática en un *grid* nos permitirá obtener resultados que, con la búsqueda aleatoria, podemos conseguir prácticamente gratis.

La implementación de la búsqueda aleatoria es prácticamente igual de simple que la de la búsqueda sistemática, paralelizable de forma trivial y, además, nos permite obtener resultados sorprendentemente buenos probando sólo un número pequeño de configuraciones de los hiperparámetros, por lo que resulta mucho más eficiente que la búsqueda sistemática. No hay motivo alguno para no utilizarla.

■ Búsqueda inteligente [smart search]

¿Podemos de alguna forma guiar el proceso de búsqueda de forma que no resulte completamente aleatorio? Sí, recurriendo a técnicas de optimización de hiperparámetros [*hyperparameter optimization*], también conocidas como técnicas de optimización basadas en modelos [*model-based hyperparameter optimization*].

En nuestra búsqueda de valores adecuados para los hiperparámetros no disponemos de un gradiente del error que guíe nuestra búsqueda como en el entrenamiento de una red, pero eso no impide que podamos construir un modelo del error observado en el conjunto de validación y utilicemos ese modelo para intentar predecir con qué valores de los hiperparámetros podemos obtener mejores resultados.

Mientras que las alternativas de buscar sistemáticamente en un

Una buena técnica de aprendizaje automático no debería ser demasiado sensible a sus hiperparámetros, algo que, por desgracia, sucede a menudo en muchas implementaciones de algoritmos de entrenamiento de redes neuronales.

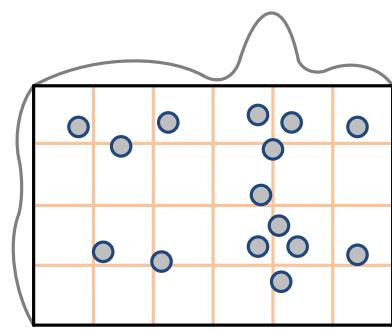


Figura 138: Ajuste de los hiperparámetros utilizando una búsqueda inteligente [smart search].

grid y de forma completamente aleatoria son “estúpidas”, al realizar una búsqueda a ciegas, podríamos decir que la optimización basada en modelos es “inteligente”, al ser una búsqueda heurística guiada.

La optimización de hiperparámetros es un área de investigación en la que se diseñan algoritmos que intentan ser más eficientes a la hora de explorar el espacio de configuraciones de los hiperparámetros de los algoritmos. Como en otras técnicas de búsqueda para problemas de optimización en espacios complejos, la clave consiste en equilibrar exploración y explotación [*exploration/exploitation trade-off*]: exploración de zonas desconocidas del espacio y explotación de las zonas donde ya hemos descubierto buenas soluciones.

Los modelos utilizados tratan de estimar tanto el valor esperado del error en el conjunto de validación para cada hiperparámetro como la incertidumbre acerca de dicha expectativa. El proceso de búsqueda consiste entonces en compaginar propuestas de hiperparámetros para los que la incertidumbre es elevada, que pueden proporcionar mejoras notables pero también peores resultados (exploración) y propuestas para las que nuestro modelo tenga confianza en que se obtendrán resultados tan buenos o mejores como los que ya conocemos, usualmente con hiperparámetros similares a los ya evaluados (explotación).

En cuanto a su implementación, la búsqueda inteligente es mucho menos paralelizable que sus alternativas no guiadas, la búsqueda en un *grid* y la búsqueda aleatoria. En lugar de generar un conjunto de candidatos y evaluarlo en paralelo, la búsqueda inteligente ha de seleccionar una configuración particular para los hiperparámetros, evaluar su calidad sobre el conjunto de validación y elegir qué hacer a continuación a partir de la información recopilada, un proceso inherentemente secuencial. Al no ser paralelizable, puede requerir más tiempo de reloj [*wall clock time*] que una búsqueda aleatoria, aunque si la técnica está bien diseñada nos permitirá reducir el número necesario de evaluaciones y ahorrar en capacidad de cálculo necesaria. Todo dependerá de nuestro presupuesto (y de la disponibilidad de un entorno en el que podamos realizar, o no, ejecuciones paralelas).

Como cualquier otra técnica de optimización, las técnicas de optimización de hiperparámetros también pueden tener sus propios parámetros, que deberemos ajustar adecuadamente (¿hiper-hiperparámetros?). En ocasiones, si no se ajustan adecuadamente esos parámetros, la búsqueda supuestamente inteligente puede obtener peores resultados que una simple búsqueda aleatoria.

Aunque a veces resulta difícil batir a un experto que es capaz de elegir cuidadosamente los intervalos en los que realizar una búsqueda aleatoria, las técnicas inteligentes pueden proporcionarnos resultados satisfactorios en ausencia de un mejor guía. Como no disponemos de una fórmula matemática para la función que deseamos optimizar, lo

que nos impide utilizar el gradiente descendente, podemos emplear diversas estrategias para resolver el problema de optimización de los hiperparámetros. Las distintas estrategias deciden de distinta forma qué configuración particular de hiperparámetros probar en cada momento. Algunas se limitan a utilizar algún criterio de tipo heurístico (p.ej. optimización sin derivadas). Otras crean un modelo de la función desconocida de error en el conjunto de validación y usan ese modelo para tomar sus decisiones (p.ej. optimización bayesiana³⁶⁷).

- Hyperopt³⁶⁸

<https://github.com/jberg/hyperopt>

Hyperopt es uno de los sistemas de optimización de hiperparámetros más conocido. Implementa tanto una búsqueda aleatoria como una búsqueda inteligente basada en un estimador de Parzen con estructura de árbol [*TPE: Tree-structured Parzen Estimator*]. Este modelo, de tipo probabilístico, se utiliza para estimar la “mejora esperada” de una configuración determinada en función de los resultados anteriores y decidir así cuál es la siguiente configuración de hiperparámetros que el algoritmo secuencial debería evaluar.

- Spearmint³⁶⁹

<https://github.com/HIPS/Spearmint>

Jasper Snoek, Hugo Larochelle y Ryan P. Adams resuelven el problema de optimización bayesiana utilizando un proceso gaussiano [*GP: Gaussian Process*]. Un proceso gaussiano especifica una distribución sobre funciones. Cuando uno muestrea un proceso gaussiano, está obteniendo una función completa. Esa función se puede utilizar para calcular la mejora esperada de diferentes configuraciones particulares de hiperparámetros (igual que TPE en Hyperopt) y guiar, de esa forma, el proceso de optimización de hiperparámetros.

- SMAC [*Sequential Model-based Algorithm Configuration*]³⁷⁰

<http://www.ml4aad.org/algorithim-configuration/smac/>

Frank Hutter, Holger H. Hoos y Kevin Leyton-Brown resuelven el problema entrenando un “bosque aleatorio” [*random forest*] de árboles de regresión para modelar la superficie de respuesta (el término técnico para denominar la función desconocida que deseamos optimizar). Un *random forest* es un tipo de ensemble que, en este caso, se utiliza para muestrear configuraciones de hiperparámetros en regiones que el ensemble considere óptimas. Hay quien dice que este método funciona mejor que los procesos gaussianos para hiperparámetros de tipo categórico.

- DNGO [*Deep Networks for Global Optimization*]³⁷¹

Al fin y al cabo, el modelo que se utiliza para guiar la optimización de los hiperparámetros es un modelo de aprendizaje automático.

³⁶⁷ David J. C. MacKay. Bayesian interpolation. *Neural Computation*, 4(3):415–447, May 1992. ISSN 0899-7667. DOI: 10.1162/neco.1992.4.3.415

³⁶⁸ James S. Bergstra, Rémi Bardenet, Yoshua Bengio, y Balázs Kégl. Algorithms for hyper-parameter optimization. En J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *NIPS’2011 Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011. URL <https://goo.gl/mBi6jg>

³⁶⁹ Jasper Snoek, Hugo Larochelle, y Ryan P Adams. Practical bayesian optimization of machine learning algorithms. En F. Pereira, C. J. C. Burges, L. Bottou, y K. Q. Weinberger, editores, *NIPS’2012 Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012. URL <https://goo.gl/dZsHjf>

³⁷⁰ Frank Hutter, Holger H. Hoos, y Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. En Carlos A. Coello Coello, editor, *LION 5: 5th International Conference on Learning and Intelligent Optimization, Rome, Italy, January 17-21, 2011. Selected Papers*, pages 507–523. Springer, Berlin, Heidelberg, 2011. ISBN 978-3-642-25566-3. DOI: 10.1007/978-3-642-25566-3_40

³⁷¹ Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat, y Ryan P. Adams. Scalable bayesian optimization using deep neural networks. En *ICML’2015: Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 6-11 July 2015*, pages 2171–2180, 2015. URL <http://jmlr.org/proceedings/papers/v37/snoek15.html>

Entonces, ¿por qué no usar una red neuronal? Es lo que Jasper Snoek y sus colaboradores del MIT, Harvard, Toronto, Intel y NERSC debieron pensar cuando propusieron emplear una red con 3 capas ocultas y unas 50 neuronas para sustituir a los métodos basados en procesos gaussianos como los de Spearmint. Aunque Spearmint puede obtener mejores resultados que TPE (Hyperopt) o SMAC, su coste computacional es elevado, de orden cúbico, y no resulta escalable, motivo por el cual se propuso DNGO, que utiliza redes neuronales para modelar distribuciones de funciones en lugar de procesos gaussianos.

- Hypergrad³⁷²

<https://github.com/HIPS/hypergrad>

Ya puestos, podemos intentar extraer gradientes con respecto a los hiperparámetros aplicando la regla de la cadena... ¡al proceso de entrenamiento completo! Esto nos permite optimizar miles de hiperparámetros: tasas de aprendizaje, momentos, parámetros de regularización y hasta la estrategia de inicialización de los pesos de la red. El uso de hipergradientes (los gradientes de los hiperparámetros), aunque curioso, presenta dificultades que hacen que no sea del todo viable. Al fin y al cabo, si la presencia de múltiples capas ocultas ya complicó el uso de *backpropagation* durante décadas, ¿qué se puede esperar si encadenamos no una, sino múltiples veces, una red neuronal multicapa? Por no hablar de qué hacer cuando los hiperparámetros son discretos, como el número de capas de una red o el de neuronas de una capa particular. En fin, una idea curiosa pero que difícilmente se llevará realmente a la práctica.

- Optimización sin derivadas³⁷³

Una estrategia más habitual (y útil) consiste en el uso de métodos libres de derivadas [*derivative-free methods*], que recurren a heurísticas para decidir qué configuración particular de hiperparámetros evaluar a continuación.

Como su propio nombre indica, se trata de técnicas de optimización para problemas en los que no se dispone de información relativa al gradiente, como el método de Nelder-Mead, que resulta sencillo de implementar y no es menos eficiente que las técnicas bayesianas de optimización.

Esencialmente, estos métodos consisten en evaluar un conjunto de puntos elegidos aleatoriamente, aproximar de alguna forma el gradiente del error y utilizar esa (burda) estimación para elegir en qué dirección movernos dentro del espacio de configuraciones de los hiperparámetros. Sin duda, mucho más práctico que el uso de hipergradientes, aunque venga mucho menos.

- Congelación y deshielo [*freeze-thaw*]³⁷⁴

Una de las limitaciones comunes de los algoritmos de optimi-

³⁷² Dougal Maclaurin, David K. Duvenaud, y Ryan P. Adams. Gradient-based hyperparameter optimization through reversible learning. En Francis R. Bach y David M. Blei, editores, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 2113–2122. JMLR.org, 2015b. URL <http://jmlr.org/proceedings/papers/v37/maclaurin15.html>

³⁷³ Andrew R. Conn, Katya Scheinberg, y Luis N. Vicente. *Introduction to Derivative-Free Optimization*. MPS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2009. ISBN 0898716683

Los algoritmos genéticos también se podrían considerar técnicas de optimización sin derivadas, aunque no resulten particularmente indicados si lo que pretendemos es minimizar el número necesario de evaluaciones.

³⁷⁴ Kevin Swersky, Jasper Snoek, y Ryan Prescott Adams. Freeze-thaw bayesian optimization. *arXiv e-prints*, arXiv:1406.3896, 2014. URL <http://arxiv.org/abs/1406.3896>

zación de hiperparámetros es que requieren entrenar una red por completo antes de extraer información con la que decidir por dónde continuar su búsqueda. Desde el punto de vista de los recursos computacionales que se desperdician, resultan más inefficientes que un proceso de optimización manual en el que un experto humano es capaz de detectar, antes de que termine, si tiene sentido continuar el experimento actual. El método de optimización bayesiana denominado “congelación-deshielo” [*freeze-thaw*] es capaz de monitorizar experimentos en curso. Esta técnica es capaz decidir cuándo lanzar un nuevo experimento, cuándo “congelar” un experimento que ya no parece demasiado prometedor y cuándo “descongelar” un experimento previamente congelado cuando resulta más prometedor tras descubrir información adicional.

- Ajuste perezoso de hiperparámetros [*lazy tuning*]³⁷⁵

Si de alguna forma fuésemos capaces de determinar cuándo una configuración determinada de hiperparámetros no proporcionará resultados prometedores, podríamos evitar su evaluación de antemano, que recordemos que resulta especialmente costosa cuando tenemos que entrenar una red neuronal completa. En cierto modo, estaríamos descubriendo la forma de “congelar” una configuración antes de que llegase a existir. Para conseguirlo, podríamos utilizar algún criterio de tipo estadístico (como un test de hipótesis) o cualquier otra regla de tipo heurístico que suela funcionar bien en la práctica.

- Google AutoML

Dada la importancia que han adquirido las técnicas de *deep learning* en la resolución de problemas prácticos reales, desde sistemas de reconocimiento de voz hasta la identificación de objetos en imágenes o el control de vehículos autónomos, y la dificultad de establecer adecuadamente su amplia gama de hiperparámetros, es normal que se hayan propuesto múltiples ideas para intentar automatizar este proceso.

Tradicionalmente, el proceso se venía haciendo a mano, confiando en la intuición del experto humano que ajustaba todos los parámetros necesarios. En empresas con múltiples proyectos relacionados con el *deep learning*, se han creado sistemas cuyo objetivo es automatizar ese proceso manual. En el caso de Google, el sistema se llama AutoML. Según Quoc Le, uno de los investigadores involucrados en el proyecto AutoML, el sistema ha sido capaz de proponer arquitecturas para resolver problemas específicos que expertos humanos no consideraban adecuadas para ese tipo de problemas. En cierto modo, un sistema de este tipo es capaz de descubrir cosas que nosotros no sabemos, lo que puede resultar sorprendente en ocasiones.

³⁷⁵ Alice X. Zheng y Mikhail Bilenko. Lazy paired hyper-parameter tuning. En Francesca Rossi, editor, *IJCAI'2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 1924–1931. IJCAI/AAAI, January 2013. ISBN 978-1-57735-633-2. URL <https://www.ijcai.org/Abstract/13/284>

Sin embargo, no lancemos las campanas al vuelo. Este tipo de técnicas resulta demasiado costoso para que su uso se generalice. En los experimentos de Google se emplearon, durante semanas, cientos de procesadores especializados TPU. Sólo la factura del consumo eléctrico asociado a tales experimentos queda fuera del alcance del común de los mortales.

Elegir adecuadamente los hiperparámetros necesarios para el entrenamiento de una red neuronal artificial es crucial para que las técnicas de *deep learning* puedan cumplir las expectativas, a veces irreales, que han creado a su alrededor. Sin embargo, no resulta ni mucho menos fácil hacerlo correctamente. Habitualmente, los resultados obtenidos dependerán de la pericia del usuario que sea capaz de intuir qué hiperparámetros resultan clave en función del contexto y el sentido en el que hay que ajustarlos. Esta habilidad se puede adquirir con la práctica.

El uso de técnicas automatizadas de optimización de los hiperparámetros de una red puede ayudar a conseguir mejores resultados. En ocasiones, técnicas automatizadas como las descritas pueden sugerir diseños de redes que rivalizan o incluso baten los diseños de los mejores expertos en *deep learning*. A menudo, todo hay que decirlo, son incapaces de encontrar configuraciones adecuadas para resolver un problema, por lo que no queda más remedio que volver al modo tradicional, el ajuste manual de todos los hiperparámetros.

Possiblemente no tenga acceso a un sistema de la escala de AutoML, pero sí puede encontrar en Internet múltiples herramientas con las que complementar sus esfuerzos a la hora de entrenar una red neuronal, automatizando parte del laborioso proceso que requiere la resolución óptima de un problema real usando técnicas de *deep learning*. Tal vez le interese analizar más detenidamente y darles una oportunidad a herramientas de optimización de hiperparámetros como Spearmint, SMAC o Hyperopt, sabiendo que aún están en fase de maduración y puede que no resulten del todo fiables en muchos aspectos.

En el futuro, no sería de extrañar que surgiessen nuevas herramientas de este tipo que requieran un uso menos intensivo de recursos computacionales. Resultarán imprescindibles si esperamos que algún día las máquinas sean capaces de aprender sin recibir instrucciones explícitas de los humanos que las controlamos. Por ahora, a la espera de mejores técnicas de optimización de hiperparámetros, el ajuste manual de los mismos suele ser necesario en la práctica.

El ajuste adecuado de los hiperparámetros de entrenamiento de una red neuronal puede marcar el éxito o el fracaso de un proyecto que se base en el uso de técnicas de *deep learning*. En el fondo, muchas competiciones de Kaggle se reducen a eso: ajustar los hiperparámetros de un algoritmo de aprendizaje automático para conseguir los mejores resultados posibles.

Google denomina TPU [*Tensor Processing Unit*] a circuitos ASIC diseñados a medida con una gran capacidad de cálculo. Su diseño no es público, aunque podemos pensar que se trata de coprocesadores matemáticos altamente paralelos que ofrecen una funcionalidad similar a la de una GPU comercial.

No es más que un problema de optimización, difícil porque no disponemos de señales explícitas que nos guíen en la búsqueda de buenas soluciones, pero un problema de optimización al fin y al cabo, que podemos intentar resolver utilizando técnicas de aprendizaje automático (meta-aprendizaje si lo prefiere). No hay razón alguna que impida que un algoritmo de aprendizaje sea capaz de aprender a ajustarse a sí mismo.

Neuroevolución

Hay quien dice que un libro nunca se termina, sólo se abandona. Lo mismo se puede decir de la optimización de redes neuronales. El espacio de configuraciones de sus hiperparámetros es tan grande que uno no termina nunca de explorar todas sus posibilidades. Simplemente, llega un momento en el que nos damos por satisfechos o se nos acaba el tiempo disponible, por lo que damos por cerrado el proceso de optimización y ponemos la red en uso.

Aunque no se usen habitualmente para resolver problemas de *deep learning*, no hay nada que nos impida utilizar técnicas de computación evolutiva para la optimización de los hiperparámetros de una red neuronal artificial, lo que da lugar a las redes neuronales evolutivas [*EANNs: Evolving Artificial Neural Networks*].³⁷⁶

El propio diseño modular de una red neuronal artificial, en el que disponemos de un sinfín de opciones a la hora de combinar bloques de distintos tipos, las hace propicias para la simulación de su evolución. Podemos ajustar el tipo, número y tamaño de sus distintas capas; las funciones de activación, los patrones de conectividad entre las capas, la existencia de enlaces recurrentes y una miríada de hiperparámetros adicionales relacionados con su proceso de entrenamiento. Dado que no existen criterios generales que podamos utilizar para tomar decisiones óptimas, los algoritmos evolutivos se pueden encargar de buscar combinaciones adecuadas que resuelvan problemas concretos de la mejor forma posible (aunque su coste computacional será, a menudo, completamente prohibitivo).

Si la mayor parte de las técnicas que hemos visto hasta ahora se centran en cómo conseguir que una red con una topología particular (una red multicapa) consiga aprender, la neuroevolución intenta descubrir cómo se puede conseguir que esas redes evolucionen. La evolución puede centrarse exclusivamente en los aspectos topológicos de la red (número de capas, interconexiones, neuronas por capa) o, también, involucrar aspectos relacionados con el comportamiento dinámico de la red.

El renacer de las redes neuronales artificiales con el *deep learning*, habilitado por la mayor disponibilidad de grandes conjuntos de datos y mayores recursos computacionales, también ha despertado el interés de algunos investigadores en la neuroevolución, cuyas aplicaciones pueden

La referencia bibliófila se refiere a que el autor nunca lo termina, sino que lo abandona. El autor siempre espera que el lector no lo encuentre interminable, aunque sea más extenso que *La historia interminable* de Michael Ende, y sí que llegue a terminarlo.

³⁷⁶ Keith L. Downing. *Intelligence Emerging: Adaptivity and Search in Evolving Neural Systems*. MIT Press, 2015. ISBN 0262029138

Una red multicapa convencional, entrenada con gradiente descendente estocástico y *backpropagation*, equivale a la implementación de una simple función no lineal. Puede que con millones de parámetros, pero simple al fin y al cabo. Sin embargo, si añadimos características adicionales, como los enlaces recurrentes de las redes recurrentes o capacidad de memoria como en las máquinas de Turing neuronales, se pueden conseguir sistemas autoorganizativos con un comportamiento dinámico mucho más rico y complejo. Además de mucho más impredecibles, menos controlables y, por qué no decirlo, muchísimo más difíciles de entrenar.

complementar a las técnicas más convencionales y ofrecen potencialmente la posibilidad de descubrir nuevas alternativas a la hora de diseñar redes neuronales artificiales (o, incluso, a la hora de entrenarlas utilizando sistemas de incentivos diferentes, no necesariamente basados en el uso de gradientes).

Algunos investigadores destacados del área son Kenneth O. Stanley, de Uber AI Labs y UCF [University of Central Florida], o Risto Miikkulainen, de la Universidad de Texas en Austin. A lo largo de los años, han propuesto numerosas alternativas para modelar la evolución de redes neuronales.

Algunas de esas alternativas se basan en codificaciones directas [*direct encodings*] de los parámetros de una red neuronal (p.ej. su topología):

- SANE [*Symbiotic Adaptive Neuroevolution*].³⁷⁷
- ESP [*Enforced Sup-Populations*].³⁷⁸
- NEAT [*NeuroEvolution of Augmenting Topologies*].³⁷⁹
- CGP [*Cartesian Genetic Programming*].^{380,381}

Otras alternativas utilizan codificaciones indirectas [*indirect encodings*], en las que los genes utilizados en la representación de las redes en los algoritmos evolutivos codifican reglas con las que luego se pueden construir las redes en sí:

- CE [*Cellular Encoding*], propuesto por Frédéric Gruau y Darrell Whitley.
- G2L [*Graph Grammar L-Systems*], de Egbert Boers e Ida Sprinkhuizen-Kuyper.
- HyperNEAT [*Hypercube-based NEAT*] y ES-HyperNEAT [*Evolvable Substrate HyperNEAT*], propuestas por Kenneth Stanley.³⁸²
- MENA [*Model of Evolving Neural Aggregates*], descrito en el libro de Keith Downing.

Todas esas propuestas resultan muy interesantes desde el punto de vista conceptual. Proponen distintas formas de combinar la trilogía de la adaptación: filogenética, ontogénesis y epigenética [*POE: phylogeny, ontogeny, and epigenesis*]. La filogenética estudia la historia evolutiva y las relaciones entre individuos y grupos de individuos como especies o poblaciones. La ontogénesis, más conocida como morfogénesis, en cuyo estudio matemático también fue pionero Alan Turing, describe el desarrollo de un organismo a lo largo de su vida. Finalmente, la epigenética hace referencia al estudio de los factores no heredables que interaccionan con los genes e intervienen en el desarrollo de un organismo, como el ambiente en el que se desarrolla su vida. En un organismo biológico, todo lo que modifica la actividad del ADN sin alterar su secuencia.

³⁷⁷ David E. Moriarty y Risto Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5(4):373–399, December 1997. ISSN 1063-6560. DOI: 10.1162/evco.1997.5.4.373

³⁷⁸ Faustino Gomez y Risto Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, January 1997. ISSN 1059-7123. DOI: 10.1177/105971239700500305

³⁷⁹ Kenneth O. Stanley y Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, June 2002. ISSN 1063-6560. DOI: 10.1162/106365602320169811

³⁸⁰ Gul Muhammad Khan, Julian F. Miller, y David M. Halliday. Evolution of Cartesian Genetic Programs for Development of Learning Neural Architecture. *Evolutionary Computation*, 19(3):469–523, 2011b. DOI: 10.1162/EVCO_a_00043

³⁸¹ Maryam Mahsal Khan, Arbab Masood Ahmad, Gul Muhammad Khan, y Julian F. Miller. Fast learning neural networks using Cartesian Genetic Programming. *Neurocomputing*, 121:274–289, 2013. DOI: 10.1016/j.neucom.2013.04.005

³⁸² Sebastian Risi y Kenneth O. Stanley. An Enhanced Hypercube-Based Encoding for Evolving the Placement, Density, and Connectivity of Neurons. *Artificial Life*, 18(4):331–363, 2012. DOI: 10.1162/ARTL_a_00071

Sin embargo, desde el punto de vista práctico, poseen una limitación común. Conforme las redes aumentan de tamaño, la escasa escalabilidad de esas técnicas hace que su rendimiento se resienta. Las aplicaciones reales de este tipo de técnicas son, por tanto, más bien reducidas, si no inexistentes, cuando hablamos de *big data* y *deep learning*. Su uso, hasta el momento, se ha limitado más a estudios sobre vida artificial [*ALife: Artificial Life*], donde se suele simular la coevolución de individuos y especies en entornos virtuales. También se pueden aprovechar, por ejemplo, en el diseño de videojuegos.

Las técnicas neuroevolutivas pueden verse como versiones con esteroides de las múltiples técnicas heurísticas de poda y crecimiento de redes neuronales artificiales que proliferaron en los años 90. Esas técnicas, poco a poco, fueron sustituidas por técnicas de regularización como las que analizaremos en el próximo capítulo.

En la página web del libro puede encontrar algunas referencias bibliográficas adicionales sobre el tema: [Entrenamiento > Hiperparámetros > Ajuste de la topología de la red](#).

Prevención del sobreaprendizaje

Desde el punto de vista formal, una red neuronal multicapa es un aproximador universal. Por este motivo, en principio, debería ser capaz de aprender cualquier cosa que nos pueda interesar. Ésa es la teoría. Sin embargo, en la práctica, no existen criterios formales que nos permitan decidir, de antemano, cuál es la configuración más adecuada de los hiperparámetros del algoritmo de entrenamiento que nos garantice un aprendizaje adecuado. De hecho, nada nos garantiza que la red sea capaz de generalizar correctamente más allá del conjunto de entrenamiento.

Aunque seamos capaces de conseguir que nuestro algoritmo de aprendizaje sea capaz de reducir al mínimo el error sobre el conjunto de entrenamiento, eso no quiere decir que estemos actuando de acuerdo a nuestros mejores intereses. Posiblemente, estemos cayendo en el problema del sobreaprendizaje, cuando un modelo funciona bien sobre su conjunto de entrenamiento pero comienza a fallar sobre datos reales que no formaron parte de su entrenamiento. Obviamente, nos gustaría conseguir modelos que funcionen bien en el conjunto de entrenamiento, como prueba de que hemos aprendido algo. Pero también que se comporten adecuadamente en los conjuntos de datos que nos encontraremos una vez puesto el modelo en uso, como prueba de que estamos generalizando correctamente (de ahí que la evaluación de los modelos la hagamos siempre con conjuntos de prueba independientes del conjunto de datos de entrenamiento).

Las técnicas de regularización nos ayudarán a reducir el error de generalización de nuestros modelos, intentando prevenir las “cicatrices” que deja un proceso de entrenamiento nunca del todo ideal.

Las “cicatrices” del entrenamiento son los efectos secundarios que causa su entrenamiento en el comportamiento posterior de un sistema. En ocasiones, pueden tener consecuencias nefastas. Es el caso de policías entrenados que, tras arrebatarle hábilmente el arma a un delincuente, instintivamente se la devuelven de forma automática, tal como solían hacer con su instructor durante su etapa de entrenamiento. Un efecto secundario indeseable que claramente nos gustaría evitar tomando las medidas necesarias, para lo cual posiblemente haya que introducir cambios en los protocolos utilizados durante su entrenamiento. De la misma forma, habrá cambios que podremos introducir en el proceso de entrenamiento de un modelo de aprendizaje automático que no están pensados para reducir el error de resustitución (sobre el conjunto de entrenamiento), sino para reducir futuros errores de generalización (que evaluamos sobre un conjunto de prueba).

Como afirmaba el economista Milton Friedman, las mejores teorías son las más simplificadas, siempre que sus predicciones sean precisas,

ya que de esa forma explican más con menos. A Albert Einstein se le atribuye una cita similar: todo debería hacerse tan simple como sea posible, pero no más. No dejan de ser distintas formas de plantear el principio filosófico de la navaja de Occam, enunciada en latín como la ley de la parsimonia (frugalidad o brevedad) por un monje franciscano del siglo XIV: “*Entia non sunt multiplicanda praeter necessitatem*” (las entidades no deben multiplicarse más allá de la necesidad). En términos más actuales, construimos modelos en los que asumimos que la explicación más sencilla es posiblemente la correcta. Para explicar un fenómeno para el que disponemos de múltiples hipótesis, nos parece razonable aceptar la hipótesis más simple, la que requiera menos supuestos no probados.

El estadístico George Box incluía un conocido aforismo en uno de sus artículos:³⁸³ “Todos los modelos están equivocados”. Posteriormente, matizaría su aforismo con un “todos los modelos están equivocados, pero algunos son útiles”.³⁸⁴ Al fin y al cabo, un modelo, como el que construimos a partir de un conjunto de entrenamiento, no deja de ser una simplificación de la realidad. Una representación idealizada de la realidad, pero no la realidad en sí misma. Ahora bien, si el modelo captura los aspectos esenciales de la realidad, puede ser muy útil en la práctica (o completamente inútil, cuando se muestra incapaz de capturarlos).

Hablar de hipótesis más sencillas y de la navaja de Occam cuando estamos entrenando redes neuronales que pueden tener millones de parámetros ajustables puede parecer un oxímoron, algo casi paradójico. De hecho, cuando disponemos de pocos datos, muchas veces podemos utilizar modelos más simples y obtener mejores resultados. Un sencillo modelo de regresión lineal o un clasificador estadístico de tipo SVM, con relativamente pocos parámetros, pueden conseguir mejores resultados que una red neuronal con varias capas ocultas... pero esos resultados posiblemente se deban a que no hemos entrenado adecuadamente la red neuronal. Una red neuronal bien entrenada, aunque dispongamos de pocos ejemplos de entrenamiento, puede batir a otros modelos. Para ello, no obstante, tendremos que recurrir a múltiples técnicas que nos ayuden a prevenir el sobreaprendizaje sobre el conjunto de entrenamiento.

El problema del sobreaprendizaje

El físico Enrico Fermi le atribuye a John von Neumann otra cita relevante: ”Con cuatro parámetros puedo ajustar un elefante, con cinco puede hacer que menee su trompa.” Si en una red neuronal tenemos millones de parámetros, casi que podemos hacer que cada elefante de los que quedan en el mundo tenga su meneo de trompa particular. En palabras del economista Ronald Coase, podemos torturar a los datos hasta que éstos confiesen. Para algunos, es de lo que trata la minería de datos [*data mining*].

³⁸³ George E. P. Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976. DOI: 10.1080/01621459.1976.10480949

³⁸⁴ George E. P. Box. Robustness in the strategy of scientific model building. En Robert L. Launer y Graham N Wilkinson, editores, *Robustness in Statistics*, pages 201–236. Academic Press, 1979. ISBN 0124381502

En realidad, se puede dibujar un elefante con sólo cuatro parámetros, si admitimos que éstos sean números complejos. Con cinco, si busca en Internet, verá que también puede mover su trompa.

Jürgen Mayer, Khaled Khairy, y Jonathan Howard. Drawing an elephant with four complex parameters. *American Journal of Physics*, 78(6):648–649, 2010. DOI: 10.1119/1.3254017

El sobreaprendizaje aparece cuando disponemos de demasiadas hipótesis válidas pero no de suficientes datos para poder descartar todas menos una. Es algo que no sólo sucede con redes neuronales de millones de parámetros. Por sencillo que sea nuestro modelo de aprendizaje automático, el temido sobreaprendizaje siempre realiza su aparición. ¿Por qué? Porque el número de hipótesis crece exponencialmente con el número de atributos. Para simplificar, si tenemos n atributos binarios, éstos dan lugar a 2^n patrones diferentes. Ahora bien, si definimos funciones sobre esos patrones, obtendremos 2^{2^n} funciones diferentes, cada una de las cuales puede describir un concepto diferente. Por tanto, el número de conceptos que un modelo podría aprender es una función exponencial de un número exponencial de patrones que se pueden obtener de un número finito de atributos. Así pues, el aprendizaje automático trata de controlar una explosión combinatoria de explosiones combinatorias.

Más formalmente, diríamos que nuestro modelo tiene una capacidad mayor que la estrictamente necesaria para generalizar correctamente, motivo por el que es capaz de aprender detalles de los datos de entrenamiento que no son realmente relevantes para la tarea para la que lo construimos. El conjunto de entrenamiento contiene patrones útiles, nadie lo duda, pero también muchos patrones que pueden resultar útiles para problemas diferentes al que pretendemos resolver. Además, también contiene ruido, que no será útil en ninguna situación: regularidades accidentales debidas al conjunto particular de datos (lo que los estadísticos denominarían error de muestreo).

Desgraciadamente, cuando ajustamos los parámetros de una red neuronal a los datos del conjunto de entrenamiento, no podemos diferenciar las regularidades realmente útiles de las irrelevantes o de las debidas al muestreo del conjunto de entrenamiento, por lo que siempre estamos expuestos al riesgo de sobreaprender [*overfitting*]. Cuando sobreaprende, un modelo se ajusta demasiado bien al conjunto de entrenamiento, modelando sus peculiaridades al detalle y su ruido. Como consecuencia, no generaliza bien: su rendimiento se degrada cuando lo utilizamos sobre casos distintos a los del conjunto de entrenamiento, que es realmente donde nos interesa que funcione bien. Una señal inequívoca de que un modelo sobreaprende es que, dotado de la capacidad suficiente, su error sobre el conjunto de entrenamiento es mucho menor que su error sobre el conjunto de prueba.

Disponemos de tres estrategias, a menudo complementarias, para evitar el sobreaprendizaje:

- *Obtener más datos.*

En *deep learning*, suele ser la mejor opción si disponemos de la capacidad de cálculo necesaria para entrenar la red utilizando un conjunto de datos de entrenamiento más grande. Ya la utilizamos cuando am-

pliamos el conjunto de datos de entrenamiento para hacer a la red robusta frente a transformaciones habituales en los datos de entrada. Y también para mejorar su robustez frente a posibles ataques usando técnicas de entrenamiento con adversario.

El sobreaprendizaje es habitual si los datos incluyen muchas características en proporción al número de observaciones disponible, algo común en determinadas aplicaciones (p.ej. análisis de datos biomédicos). En tales situaciones, los conjuntos de datos son más anchos que altos y no siempre podremos obtener más datos con facilidad. A menudo se reduce la dimensionalidad de los datos seleccionando o extrayendo características, aunque entonces se corre el riesgo de estar desperdiando información potencialmente útil. En cualquier caso, estaríamos hablando ya de la segunda de las estrategias de las que disponemos para evitar el sobreaprendizaje:

- *Ajustar la capacidad del modelo.*

Ya analizamos cómo muchos detalles del entrenamiento de la red son de vital importancia para conseguir un rendimiento adecuado, desde las funciones de activación, la topología de la red y la inicialización de sus pesos hasta las técnicas de ajuste dinámico de las tasas de aprendizaje que facilitan la convergencia del gradiente descendente. Los modelos de *deep learning* tienen múltiples matrices y requieren que seamos cuidadosos a la hora de establecer sus hiperparámetros, tal vez con la ayuda de costosos procesos de búsqueda que los ajusten automáticamente.

Ajustar adecuadamente los parámetros de la red también nos puede servir para que la red tenga la capacidad adecuada. Suficiente para identificar las regularidades relevantes en los datos de entrenamiento, pero no demasiada para que también se ajuste a las espurias. Obviamente, estamos suponiendo que las espurias serán más débiles que las auténticas, ya que de otro modo el aprendizaje resultaría por completo imposible.

- *Combinar múltiples modelos.*

Una tercera estrategia para evitar el sobreaprendizaje consiste en asumir que resulta inevitable y combinar múltiples modelos con la intención de que los errores cometidos por algunos de ellos se compensen con los aciertos de los demás. Cuando hablamos de redes neuronales, esta combinación se puede realizar de dos formas diferentes:

- Promediando múltiples modelos [*model averaging*]: Se construyen muchos modelos diferentes con diferentes parámetros o el mismo tipo de modelo utilizando distintos subconjuntos del conjunto de entrenamiento (p.ej. *bagging*). Esto es, se crea un ensemble, tal como haríamos con cualquier otra técnica de aprendizaje automático.

- Aplicando un enfoque bayesiano [*Bayesian fitting*], se selecciona una única arquitectura de red. A continuación, se combinan las predicciones realizadas por muchos vectores de pesos diferentes, que se obtienen tras repetir el entrenamiento de la red en múltiples ocasiones. Esto es, en vez de entrenar la red con distintos subconjuntos de datos de entrenamiento, como haríamos en *bagging*, aprovechamos la naturaleza estocástica del algoritmo de entrenamiento de redes neuronales para entrenar múltiples versiones de una misma red neuronal.

En el fondo, todo se reduce al problema del sesgo y la varianza [*bias/variance trade-off*]. Si nuestro modelo no tuviese la capacidad suficiente para aprender del conjunto de datos, su rendimiento nunca podría ser óptimo debido a su sesgo. El modelo resultaría demasiado simple y hasta dogmático, como aquel que sólo dispone de un martillo y para él todo son clavos. Cuando nuestro modelo es bastante flexible y tiene más capacidad de la estrictamente necesaria, como sucede con una red neuronal lo suficientemente grande, su rendimiento no será el más adecuado debido a la varianza. El sobreaprendizaje, por tanto, aparece cuando la varianza es elevada y, como siempre, el secreto está en llegar a un equilibrio entre el sesgo y la varianza para que el rendimiento del modelo resulte óptimo.

Las redes neuronales, como ya sabemos, disponen de montones de parámetros, los pesos sinápticos que determinan las interacciones de unas neuronas con otras. Como consecuencia, son modelos especialmente propensos a exhibir una elevada varianza y ser víctimas del sobreaprendizaje. Por suerte para nosotros, los investigadores del tema son perfectamente conscientes del problema y, a lo largo de los años, han propuesto multitud de técnicas para combatir el sobreaprendizaje, que también podríamos denominar técnicas de reducción de la varianza. A menudo, son técnicas específicamente diseñadas para su uso en redes neuronales, no aplicables a otros modelos de aprendizaje automático. Por ejemplo, una de las más populares, denominada *dropout*, podríamos decir que funciona con el gradiente descendente estocástico como el uso de *bagging* para construir ensambles, salvo que en vez de muestrear el conjunto de entrenamiento se muestrean los parámetros de la red (sus pesos).

Las técnicas de reducción de la varianza nos permiten entrenar modelos enormes, con multitud de parámetros incluso aunque no dispongamos de conjuntos de datos igual de enormes, casi un anatema para un estadístico. Las redes neuronales, gracias a su modularidad, nos permiten incorporar restricciones específicas del problema que pretendemos resolver (p.ej. invarianza frente a rotaciones y traslaciones). Esas restricciones impuestas sobre el modelo nos ayudan a reducir su varianza. Usando la terminología bayesiana, a esas restricciones impuestas mediante las cuales reducimos

En aprendizaje automático, los dogmas son más comunes de lo que pueda parecer. Hay quien, en vez de buscar la técnica más efectiva para resolver un problema, parece empeñado en resolver cualquier problema usando ‘su’ técnica. Algo tan absurdo, en principio, como intentar ajustar una línea recta sobre datos no lineales (el ejemplo típico de modelo sesgado).

La asociación tradicional entre *deep learning* y *big data*, según la cual sólo cuando disponemos de enormes cantidades de datos podemos sacarle partido a lo que nos ofrecen las técnicas de *deep learning*, no se traduce, ni mucho menos, en que no podamos utilizar *deep learning* cuando trabajemos con conjuntos de datos más modestos. Se trata sólo de un mito que hay que desterrar. Las técnicas de reducción de la varianza nos permitirán entrenar redes neuronales aun cuando la proporción entre parámetros de la red y datos de entrenamiento sea elevada, una clara señal del riesgo inequívoco de sobreaprendizaje, que tendremos que aprender a prevenir o, al menos, mitigar parcialmente.

drásticamente la varianza del modelo se las suele denominar *priors* (término derivado de las probabilidades a priori, $P(A)$, que utilizamos para representar nuestro grado de creencia inicial en A a la hora de calcular $P(A|B)$ usando el teorema de Bayes).

Esas restricciones, o *priors* si lo prefiere, sirven para sesgar nuestro modelo a favor de las propiedades que creemos que debería tener. Si nuestra creencia tiene cierta base real, el incremento del sesgo y la reducción de la varianza que se consigue nos permitirá mejorar la capacidad predictiva del modelo que construyamos. Gracias a ellas, no hace falta que dispongamos de los datos que tienen gigantes como Google, Facebook o Microsoft para usar con éxito técnicas de *deep learning*. Incluso aunque dispusiésemos de ellos, las técnicas de reducción de la varianza nos seguirían siendo extremadamente útiles. Estas técnicas son las que nos ayudarán a prevenir el sobreaprendizaje en el entrenamiento de redes neuronales.

Sólo existe una situación en la que el sobreaprendizaje no deba preocuparnos en exceso. Cuando se dispone de conjuntos de datos que crecen más rápido que nuestra capacidad para procesarlos, no queda más remedio que utilizar cada ejemplo de entrenamiento una sola vez o, incluso, terminar el entrenamiento de la red antes de que se haya recorrido el conjunto de datos completo. El sesgo o la falta de capacidad de nuestro modelo [*underfitting*], junto con la eficiencia computacional de su entrenamiento, se convierten entonces en nuestras principales preocupaciones.³⁸⁵

Regularización

En aprendizaje automático, la regularización es cualquier modificación que realizamos sobre un algoritmo de aprendizaje con la intención de reducir su error de generalización (sobre el conjunto de prueba) pero no su error de resustitución (sobre el conjunto de entrenamiento). Es decir, el objetivo de la regularización es prevenir el sobreaprendizaje.

Recordemos, de cuando estudiamos la descomposición del error en sesgo y varianza, que un modelo de aprendizaje automático puede encontrarse en tres situaciones diferentes. La primera, cuando sufre de sesgo, impide que el modelo sea capaz de representar correctamente el proceso real que da lugar a los datos observados, al no ser el modelo lo suficientemente flexible. La segunda, que es la que perseguimos, adapta el modelo al proceso real del que se obtienen los datos, obteniendo resultados óptimos. La tercera, en la que el error se debe principalmente a la varianza, aparece cuando el modelo es demasiado flexible y se adapta demasiado bien al conjunto de datos de entrenamiento. El objetivo de la regularización es llevar un modelo del tercer escenario al segundo.

Ya sabemos, por el teorema de Wolpert, que nada es gratis: no hay un algoritmo de aprendizaje que sea consistentemente mejor que otro. De la

³⁸⁵ Léon Bottou y Olivier Bousquet. The tradeoffs of large scale learning. En *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pages 161–168, USA, 2007. Curran Associates Inc. ISBN 978-1-60560-352-0. URL <https://goo.gl/CSz6jZ>

misma forma, tampoco nos encontraremos una técnica de regularización que sea siempre preferible a las demás. Tendremos que elegir, para cada situación, una forma de regularización que se adapte bien al problema particular que deseemos resolver. Afortunadamente, no tendremos que partir de cero, sino que existe un conjunto bastante amplio de técnicas genéricas de regularización que resultan aplicables en una amplia gama de problemas.

Aunque, en principio, podría decirse que la regularización abarca cualquier proceso de introducción de información adicional en el algoritmo de entrenamiento que nos permita prevenir el sobreaprendizaje, aquí nos limitaremos a técnicas que, en el fondo, lo que hacen es ajustar la capacidad del modelo. Algunas técnicas de ampliación del conjunto de entrenamiento ya las analizamos en el capítulo anterior, dedicado al entrenamiento de redes neuronales, mientras que la combinación de múltiples modelos, usando ensambles, la consideraremos de nuevo al final de este mismo capítulo.

Uno podría pensar, ingenuamente, que, si elegimos un tipo de modelo adecuado para el problema particular que nos ocupe, entonces no nos tendríamos que preocupar de regularizarlo. Sin embargo, en la práctica, no disponemos de la capacidad que nos permita determinar cuál es ese modelo “adecuado” y, aun cuando dispusiésemos de ella, puede que no existiese ningún modelo realmente preciso para nuestro objetivo que no involucrase realizar una simulación completa del Universo desde sus orígenes.

Controlar la capacidad del modelo no es simplemente descubrir de qué tamaño debe ser la red neuronal, de cuántas capas debe constar, cuántos parámetros hemos de ajustar o con qué hiperparámetros hemos de entrenarla. Para encontrar el mejor modelo, en el sentido de minimizar su error de generalización, debemos entrenar un modelo que sea lo suficientemente grande para evitar posibles sesgos y que haya sido debidamente regularizado.

¿Qué hace que un modelo de red neuronal sea mejor que otro? Desde el punto de vista ideal, que permita identificar las causas reales que ocasionan las variaciones observadas en los datos, sus “factores causales”, una cuestión que da para discusiones interminables de tipo filosófico. En la práctica, los factores causales difícilmente los podremos identificar, pero sí que podemos utilizar algunas pistas que nos ayuden a formular hipótesis consistentes con los datos observados. En problemas de aprendizaje supervisado, las etiquetas del conjunto de entrenamiento nos servirán de guía, mediante la definición de funciones de error, coste o pérdida que utilizaremos para ajustar los parámetros de la red. En problemas de aprendizaje no supervisado, las pistas son más indirectas: creencias previas que imponemos para guiar el aprendizaje. En cualquier caso, esas guías, aunque fundamentales en el proceso de aprendizaje sobre el

Geoffrey Hinton va más allá y dice que, si la red neuronal no sobreaprende (o, más bien, cuando no tiene la capacidad suficiente como para sobreaprender), entonces no se está haciendo *deep learning*.

conjunto de entrenamiento, no nos aportan nada en cuanto al error de generalización del modelo.

¿De qué estrategias generales disponemos si deseamos ajustar la capacidad efectiva de una red neuronal mediante técnicas de regularización? Esencialmente, de las dos siguientes:

- Añadir restricciones adicionales al modelo de red neuronal, en forma de restricciones sobre los valores de sus parámetros. Por ejemplo, podemos limitar la norma del vector de pesos asociado a cada neurona particular u obligar a que pesos correspondientes a distintas sinapsis comparten siempre el mismo valor (esto es, compartiendo parámetros entre diferentes neuronas o capas de la red).
- Añadir términos extra en la función de coste o pérdida que se utiliza como objetivo en el proceso de optimización de los parámetros de la red. Esta estrategia puede verse como una forma de imponer restricciones débiles [*soft constraints*] sobre los valores de los parámetros de la red neuronal, ya que nos permite establecer, de forma indirecta, limitaciones sobre las configuraciones posibles de la red.

Las dos estrategias anteriores, utilizadas cuidadosamente, pueden ayudarnos a mejorar el rendimiento de una red neuronal sobre el conjunto de prueba (que, recordemos una vez más, no ve la red durante su entrenamiento). Comencemos por la segunda de ellas.

Regularización de la función de coste

La forma más común de regularizar el entrenamiento de una red neuronal artificial consiste en añadir términos adicionales a la función de coste o pérdida que intentamos optimizar ajustando sus parámetros. Hasta ahora, esa función de coste era, básicamente, una función de error E que evaluábamos sobre el conjunto de entrenamiento con la intención de conseguir que la red aprendiese. Al introducir un término de regularización, la función objetivo regularizada pasa a ser de la forma:

$$L = L_E + \lambda L_R$$

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

En entrenamiento de la red, como siempre, se hará calculando el gradiente de la función de coste o pérdida L [*loss function*]:

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

El resto del proceso de entrenamiento de la red neuronal sigue siendo exactamente el mismo. Sólo cambia la señal que se propaga hacia atrás

cuando usamos *backpropagation*. En esa señal, además del error sobre el conjunto de entrenamiento, se incluye información que, idealmente, contribuirá a reducir el error de generalización de la red neuronal una vez entrenada.

Observe que, al incluir un término adicional en la función de coste, también hemos añadido un hiperparámetro más al algoritmo de entrenamiento de la red. El hiperparámetro de regularización λ nos permite controlar la capacidad de la red. Indirectamente, determina cómo de flexible será la red, reduciendo sus grados de libertad a la hora de ajustarse a los datos de entrenamiento y, de esa forma, contribuyendo a reducir el sobreaprendizaje. Como sucede con otros hiperparámetros involucrados en el entrenamiento de la red, establecer un valor adecuado para el parámetro de regularización λ es esencial para controlar de forma efectiva la capacidad de la red, previniendo el sobreaprendizaje y facilitando su capacidad de generalización. El parámetro λ , por tanto, ha de incluirse también entre los hiperparámetros que han de ajustarse cuando entrenamos una red neuronal (preferiblemente, utilizando técnicas automáticas como las ya vistas en capítulos anteriores).

La regularización de la función de coste tiene su origen en los métodos de contracción [*shrinkage methods*] que se emplean en Estadística para reducir la varianza de un estimador de mínimos cuadrados por medio de la inclusión de restricciones sobre los valores de los coeficientes. En las técnicas estadísticas de regresión, como en cualquier técnica de aprendizaje automático, se observa que el error de generalización (sobre el conjunto de test) es habitualmente mayor que el error de resustitución (sobre el conjunto de entrenamiento). El nombre de los métodos de contracción proviene del hecho de que, añadiendo restricciones, se consigue contraer o reducir el coeficiente de determinación de forma artificial, donde el coeficiente de determinación R^2 se define como:

$$R^2 = 1 - \frac{SSE}{SST}$$

donde SST es la suma total de los cuadrados (proporcional a la varianza de los datos) y SSE es la suma de los errores de predicción (residuos) al cuadrado.

$$\begin{aligned} SST &= \sum (y_i - \bar{y})^2 \\ SSE &= \sum (y_i - \hat{y}_i)^2 = \sum e_i^2 \end{aligned}$$

En regresión, también se usa la suma “explicada” de los cuadrados, o suma de los cuadrados debidos a la regresión, $SSR = \sum (\hat{y}_i - \bar{y})^2$. En el caso particular de la regresión lineal, se verifica la descomposición $SST = SSR + SSE$.

Así pues, el coeficiente de determinación R^2 está relacionado directamente con la fracción no explicada de la varianza [*FVU: fraction of variance unexplained*]: $R^2 = 1 - FVU$. El coeficiente R^2 es la proporción de la varianza en la variable dependiente que se puede predecir a partir de las variables independientes, mientras que *FVU* es la fracción de la

varianza que no puede predecirse a partir de las variables independientes. El coeficiente R^2 será 0 (o, FVU será 1) cuando las variables independientes no nos indiquen nada acerca de la variable dependiente y , en el sentido de que las predicciones \hat{y} no covariarán con y . En teoría, por tanto, cuanto mayor sea R^2 , mejor se ajusta el modelo de regresión a los datos, corriendo el riesgo del sobreaprendizaje, denominado sobreajuste por los estadísticos. Para reducir este sobreajuste, de forma artificial se reduce R^2 con la intención de que el modelo generalice mejor. Como en el caso general, se pretende reducir la varianza debida a los errores de muestreo.

La regresión lineal (o no lineal) que se suele aprender en cursos básicos de Estadística no suele incluir términos de regularización, entre otros motivos porque no resultan necesarios para situaciones simples en las que apenas trabajamos con unas cuantas variables; esto es, cuando la variable dependiente sólo depende de unos pocos parámetros. Sin embargo, cuando se trabaja con miles de variables, la regularización se hace imprescindible en los modelos de regresión. Los dos métodos más comunes de regresión regularizada, la regularización de Tikhonov y el método LASSO, se pueden emplear para regularizar redes neuronales artificiales, donde se conocen con el nombre de regularización L_2 y regularización L_1 , respectivamente.

A diferencia de las técnicas de reducción de la dimensionalidad, las técnicas de regularización mantienen todas las variables. Sin embargo, reducen el efecto de algunas de ellas sobre la salida del modelo. Esta reducción suele conseguirse penalizando los coeficientes asociados a dichas variables. Cuanto mayores sean los coeficientes, mayor será la penalización, lo que conduce a modelos en los que la magnitud de los coeficientes se reduce. En el caso de las redes neuronales, los parámetros ajustados son los pesos sinápticos y la reducción de su magnitud contribuye, además, a prevenir fenómenos como la saturación prematura, la cual dificulta el proceso de entrenamiento de una red.

La diferencia entre la regularización L_1 y la regularización L_2 reside en la forma que toma el término de penalización L_R en la función de pérdida utilizada para entrenar la red:

- *Regularización L_2 (a.k.a. weight decay)*

$$L = E + \frac{1}{2}\lambda \sum w_k^2$$

La regularización L_2 es la forma más habitual de regularización en redes neuronales artificiales, donde se suele conocer con el término de *weight decay* (decaimiento de pesos). En Estadística, se conoce con el nombre de regularización de Tikhonov o regresión de “arista” [*ridge regression*] cuando se aplica a la regresión por mínimos cuadrados. ¿Por qué L_2 ? Porque la norma L_2 de un vector w es su norma euclídea, $\|w\|_2$, la cual se usa, por ejemplo, para medir distancias en un espacio

euclídeo.

$$L2 = \|w\|_2^2 = \sum w_k^2$$

Al calcular el gradiente de la función regularizada para utilizarlo en el gradiente descendente con *backpropagation*, la actualización de los pesos se realiza de la siguiente manera:

$$\Delta w_k = -\eta \frac{\partial L}{\partial w_k} = -\eta \left(\frac{\partial E}{\partial w_k} + \lambda w_k \right)$$

donde se ve cómo el único cambio que se introduce en la regla de aprendizaje es un término adicional que contrae los pesos en un factor $-\eta\lambda$ en cada paso del entrenamiento de la red.³⁸⁶ Obsérvese que el factor 1/2 se introduce en el término de regularización de la función de pérdida L sólo para que el 2 desaparezca al calcular la derivada y no haya que ir arrastrándolo en todos los cálculos. De esta forma, el gradiente del término de regularización con respecto al vector de pesos w es, simplemente, λw .

El origen de esta técnica se encuentra en los años 60, cuando el matemático ruso Andrey Tikhonov la propuso como una forma de aplicar la navaja de Occam: introduciendo un término adicional que penaliza las soluciones más complejas, los modelos más complejos no sólo deben ser mejores, sino significativamente mejores que sus alternativas para justificar su mayor complejidad.³⁸⁷

La consecuencia de la regularización L2 es que el vector de pesos se acerca al origen, al incluir un pequeño factor que, en cada actualización, disminuye la magnitud de los pesos. Dada la tendencia al sobreaprendizaje de las redes neuronales, este sencillo mecanismo proporciona una forma de hacer que los pesos tiendan a desaparecer salvo que su valor se refuerce como parte de la actualización debida al gradiente del error con respecto a los pesos. Al final del entrenamiento, sólo los pesos más relevantes para reducir el error tendrán una magnitud significativamente distinta de cero. La regularización L2, de esta manera, penaliza fuertemente los pesos grandes y prima la obtención de vectores de pesos con valores pequeños. Ayuda a que la red aproveche todos sus parámetros sin darle demasiada prominencia a parámetros individuales, salvo que su importancia a la hora de reducir significativamente el error así lo justifique.

La regularización L2 se ha venido utilizando tradicionalmente en el entrenamiento de redes neuronales desde los años 90.³⁸⁸ Resultados experimentales de esa época sugirieron que el uso de regularización L2, ya denominada *weight decay*, puede hacer más robusta a la red ante la presencia de ruido en los datos.³⁸⁹

Desde un punto de vista formal, la regularización L2 hace que el algoritmo de entrenamiento de una red neuronal perciba los datos de

³⁸⁶ Geoffrey E. Hinton. Connectivist learning procedures. *Artificial Intelligence*, 40(1):185–234, 1989. ISSN 0004-3702. doi: 10.1016/0004-3702(89)90049-0

³⁸⁷ Andrey Nikolaevich Tikhonov y Vasilii Yakovlevich Arsenin. *Solutions of Ill-Posed Problems*. Scripta Series in Mathematics. V.H. Winston & Sons, 1977. ISBN 0470991240

³⁸⁸ Andreas S. Weigend, David E. Rumelhart, y Bernardo A. Huberman. Generalization by weight-elimination with application to forecasting. En *NIPS'1990 Advances in Neural Information Processing Systems 3*, pages 875–882. Morgan-Kaufmann, 1990. URL <https://goo.gl/864Mto>

³⁸⁹ Amit Gupta y Siuwa M. Lam. Weight decay backpropagation for noisy data. *Neural Networks*, 11(6):1127 – 1138, 1998. ISSN 0893-6080. doi: 10.1016/S0893-6080(98)00046-X

entrada como si tuviesen una mayor varianza de la que realmente tienen, lo que empuja al algoritmo de aprendizaje a reducir la magnitud de los pesos correspondientes a características cuya covarianza con la función objetivo es baja en comparación con esa varianza añadida. Sólo se preservan relativamente intactos los pesos que contribuyen significativamente a reducir el valor de la función de coste que pretendemos minimizar.

La idea de la regularización L2, o *weight decay*, no se utiliza únicamente en la resolución de problemas de regresión regularizada o en el entrenamiento de redes neuronales, sino que tiene aplicaciones adicionales:

- El algoritmo RLS [*Recursive Least Squares*], utilizado en la construcción de filtros adaptativos para el procesamiento de señales, es, implícitamente, un algoritmo basado en *weight decay*, aunque en su formulación original en un trabajo de Carl Friedrich Gauss de 1821, obviamente, no se planteaba como tal.³⁹⁰ Un filtro RLS minimiza una función de coste de la forma $\sum \lambda^{n-i} e^2(i)$, en la que el factor de olvido λ le da menor peso a errores más antiguos, frente a los filtros LMS [*Least Mean Squares*], cuyo objetivo es reducir el error cuadrático medio y se entrena con la regla delta de Widrow y Hoff.
- La resolución de un sistema de ecuaciones o el análisis de componentes principales PCA requieren la inversión de una matriz. Obviamente, la matriz no se puede invertir si es singular. En un sistema de ecuaciones lineales, esto ocurre cuando el sistema de ecuaciones es compatible indeterminado; esto es, admite más de una solución. En PCA, donde hay que invertir la matriz $X^\top X$, sucede cuando no existe varianza en los datos en alguna dirección (o, al menos, no se observa que exista esa varianza en los datos disponibles). Para solucionar el problema, se invierte la matriz regularizada $X^\top X + \alpha I$, que garantiza tener inversa. Una definición de la matriz pseudoinversa de Moore-Penrose, propuesta independientemente por E.H. Moore en 1920, Arne Bjerhammar en 1951 y Roger Penrose en 1955, es la siguiente:

$$X^+ = \lim_{\alpha \rightarrow 0} (X^\top X + \alpha I)^{-1}$$

que puede verse como una forma de realizar una regresión lineal con regularización L2 o *weight decay*.

La regularización permite estabilizar la resolución de problemas matemáticos indeterminados, garantizando la convergencia de los métodos iterativos de optimización utilizados para resolverlos.

■ Regularización L1

$$L = L_E + \lambda \sum |w_k|$$

³⁹⁰ Chi-Sing Leung, Kwok-Wo Wong, Pui-Fai Sum, y Lai-Wan Chan. A pruning method for the recursive least squared algorithm. *Neural Networks*, 14(2): 147 – 174, 2001. ISSN 0893-6080. DOI: 10.1016/S0893-6080(00)00093-9

El cálculo de la pseudoinversa no se realiza aplicando esta fórmula, sino una descomposición de la matriz. En un sistema de ecuaciones $Ax = y$, cuando A tiene más columnas que filas, la pseudoinversa proporciona una solución $x = A^+y$ de las muchas soluciones posibles (la que minimiza la norma euclídea $\|x\|_2$). Cuando tiene más filas que columnas, puede que el sistema no tenga solución, caso en el que la pseudoinversa nos da el valor de x para el que Ax está tan cerca de y como sea posible, en términos de la norma euclídea $\|Ax - y\|_2$.

La regularización L1 se denomina así por utilizar la norma L1, que es la que se emplea, por ejemplo, para calcular la distancia de Manhattan (la distancia que habría que recorrer, en una ciudad con un plano perfectamente cuadriculado, para llegar de un punto a otro).

$$L1 = ||w||_1 = \sum |w_k|$$

Al utilizar este término de regularización, el gradiente de la función de pérdida hace que nuestra regla de aprendizaje para actualizar los pesos usando el gradiente descendente tenga la forma:

$$\Delta w_k = -\eta \frac{\partial L}{\partial w_k} = -\eta \left(\frac{\partial E}{\partial w_k} + \lambda sign(w_k) \right)$$

donde $sign(w_k)$ es la función signo aplicada al peso w_k . Cuando el peso es positivo, su valor se reducirá; cuando el peso es negativo, su valor se incrementará. Tanto en un sentido como en otro, el cambio en el peso debido al término de regularización será de magnitud $\eta\lambda$. La contribución del término de regularización al gradiente ya no depende linealmente de cada peso w_k , como en *weight decay*, sino que es constante. El gradiente del término de regularización con respecto al vector de pesos w es, simplemente, $sign(w)$.

Originalmente, este tipo de regularización se propuso como una forma de dotar a las redes neuronales de la capacidad de olvidar.³⁹¹ Se observó que, de esta forma, las redes neuronales obtenidas eran algo menos sensibles a su configuración inicial.³⁹²

En Estadística, se atribuye la invención del método a Robert Tibshirani, de la Universidad de Stanford.³⁹³ Tibshirani, que publicó su trabajo un año después que Ishikawa, lo denominó LASSO [*Least Absolute Shrinkage and Selection Operator*], término que a menudo aparece en minúsculas, como *lasso*. Básicamente, Tibshirani integró una regularización L1 en un modelo lineal usando mínimos cuadrados, si bien el método original se puede generalizar fácilmente a otros muchos modelos estadísticos.

Como sucedía con la regularización L2, la regularización L1 tiende a reducir los valores de los parámetros de un modelo, ya estemos resolviendo un problema de regresión regularizada o entrenando una red neuronal. Además, la presión hacia abajo que realiza sobre los parámetros hace que muchos de ellos acaben siendo cero. De ahí que Tibshirani lo denominase “operador de selección”, al servir como método automático de selección de características. Como en la regularización L2, usando la regularización L1 hacemos que sólo se conserven aquellos parámetros que tienen un impacto notable sobre la función objetivo que pretendemos minimizar.

A decir verdad, tampoco fue Ishikawa el primero en proponer la regularización L1. En procesamiento de señales, se conoce como

En realidad, en Manhattan tenemos la avenida Broadway, que recorre la isla en diagonal, por lo que la distancia de Manhattan no es del todo adecuada ni siquiera en Manhattan. Aun así, hay muchos problemas en los que sí se puede utilizar con éxito.

³⁹¹ Masumi Ishikawa. Learning of modular structured networks. *Artificial Intelligence*, 75(1):51 – 62, 1995. ISSN 0004-3702. doi: 10.1016/0004-3702(94)00061-5

³⁹² R. Kozma, M. Sakuma, Y. Yokoyama, y M. Kitamura. On the accuracy of mapping by neural networks trained by backpropagation with forgetting. *Neurocomputing*, 13(2):295 – 311, 1996. ISSN 0925-2312. doi: 10.1016/0925-2312(95)00094-1

³⁹³ Robert Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society (Series B: Methodological)*, 58(1): 267–288, 1996. URL <http://www.jstor.org/stable/2346178>

eliminación de ruido en la búsqueda de bases [*BPDN: Basis Pursuit Denoising*] y la presentó Scott Shaobing Chen como parte de su tesis doctoral de 1994, dirigida por Dave Donoho, también de la Universidad de Stanford. Un ejemplo más de idea a la que le había llegado su momento y que, en un intervalo de apenas dos años, apareció en múltiples ámbitos de forma (aparentemente) independiente. La búsqueda de bases es un problema de optimización matemática de la forma $\min\|x\|_1$ sujeto a la restricción $y = Ax$, donde x es un vector solución (correspondiente a una señal), y es un vector de observaciones y A es una matriz de transformación con menos filas que columnas. Entonces, $y = Ax$ es un sistema indeterminado de ecuaciones lineales del que nos interesa su solución más dispersa (con un mayor número de ceros). Cuando estamos dispuestos a sacrificar la congruencia exacta de Ax con y a cambio de obtener un vector x más disperso, obtenemos BPDN, en el que el problema de optimización toma la forma $\min_x \frac{1}{2}\|y - Ax\|^2 + \lambda\|x\|_1$. El parámetro λ controla el compromiso entre la dispersión de la solución x y la fidelidad de la reconstrucción Ax . Se trata de un problema de optimización convexa que se puede resolver utilizando técnicas de programación cuadrática. De hecho, se han propuesto múltiples técnicas especializadas para resolver este tipo de problemas con soluciones dispersas, como el “algoritmo de la multitud” [*in-crowd algorithm*].³⁹⁴ ¿Para qué sirve todo esto? Para descomponer una señal en una superposición “óptima” de elementos (bases), lo que nos puede servir para comprimirla (p.ej. como técnica de compresión de imágenes o de los datos recibidos por un sensor).³⁹⁵

Así pues, ya sea en Estadística, en procesamiento digital de señales o en redes neuronales, la regularización L1 nos permite llegar a un compromiso entre tener un error residual pequeño y hacer el vector solución simple en términos de su norma L1. Es una forma más de darle una formulación matemática a la navaja de Occam: encontrar la explicación posible más sencilla (la que minimiza $\|w\|_1$) capaz de ser consistente con nuestras observaciones (minimizando el error E).

Ahora bien, si lo que queremos es una explicación simple, seguramente estaríamos interesados en utilizar una técnica que nos permitiese anular cuantos parámetros de nuestro modelo no fuesen estrictamente necesarios. Haciendo que muchos parámetros de nuestro modelo tomasen el valor 0, los podríamos eliminar, simplificándolo al máximo y mejorando potencialmente su interpretabilidad.

Desde este punto de vista, nos gustaría poder regularizar nuestro modelo utilizando la norma L0 de nuestro vector de parámetros, $\|w\|_0$, definida como el número de elementos distintos de cero en el vector w . Por desgracia, el problema de aprendizaje regularizado con L0 es un problema NP-difícil.³⁹⁶

³⁹⁴ Patrick R. Gill, Albert Wang, y Aljosha Molnar. The in-crowd algorithm for fast basis pursuit denoising. *IEEE Transactions on Signal Processing*, 59(10):4595–4605, Oct 2011. ISSN 1053-587X. DOI: 10.1109/TSP.2011.2161292

³⁹⁵ Scott Shaobing Chen, David L. Donoho, y Michael A. Saunders. Atomic decomposition by basis pursuit. *SIAM Journal on Scientific Computing*, 20(1):33–61, December 1998. ISSN 1064-8275. DOI: 10.1137/S1064827596304010

³⁹⁶ B. K. Natarajan. Sparse approximate solutions to linear systems. *SIAM Journal on Computing*, 24(2):227–234, 1995. DOI: 10.1137/S0097539792240406

La norma L1 podemos, no obstante, utilizarla para aproximar la solución óptima al problema de regularización con la norma L0, interpretando la regularización L1 como una relajación de la regularización que realmente nos gustaría aplicar (la basada en la norma L0). Desde esta perspectiva, la regularización L1 promueve que la solución que obtengamos sea más dispersa que la que obtendríamos con la regularización L2. Algunos de los parámetros tendrán un valor igual a cero, lo que nos permitirá eliminarlos. De esta forma, se reduce el tamaño de los modelos obtenidos, lo que se traduce en una poda de la red neuronal. La poda conseguida será mayor cuanto mayor sea el valor del parámetro de regularización λ .

La regularización con L1 nos ayudará a obtener soluciones más dispersas cuando entrenemos una red neuronal artificial. Este resultado resulta atractivo si nos fijamos en el comportamiento real de las redes neuronales biológicas, cuyos niveles de activación son extremadamente dispersos para minimizar la energía consumida por la red. La regularización L1 impone una restricción que contribuye en ese sentido, aunque claramente no tiene nada que ver con el funcionamiento real de una red neuronal de tipo biológico (no parece que sea biológicamente plausible). Se trata, no obstante, de una afortunada coincidencia.

Tanto la regularización L1 como la regularización L2 hacen que la magnitud de los pesos se reduzca. Ambas penalizan, de forma ligeramente distinta, la aparición de pesos grandes. Sin embargo, la forma en la que se reducen los pesos cambia de un método a otro. En la regularización L1, los pesos encogen una cantidad constante. Por su parte, en la regularización L2, la reducción es proporcional a valor de los pesos. Cuando la magnitud de un peso es elevada, la regularización L2 reduce el peso mucho más rápidamente que la regularización L1. Sin embargo, cuando el peso ya es pequeño, la regularización L1 es más pronunciada que la regularización L2. Al final, la regularización L1 tiende a concentrar los pesos de la red en un número relativamente pequeño de sinapsis importantes mientras que los demás pesos acaban valiendo cero. En otras palabras, la red utiliza sólo un subconjunto de sus parámetros y es indiferente a los niveles de activación de sinapsis descartadas. En cambio, la regularización L2 tiende a obtener vectores finales de pesos en los que todos los pesos tienen valores pequeños. ¿Podríamos conseguir la rápida eliminación de pesos grandes, propia de la regularización L2, con las soluciones dispersas características de la regularización L1?

- *Regularización de redes elásticas*

$$L = E + \lambda_1 \sum |w_k| + \frac{1}{2} \lambda_2 \sum w_k^2$$

Las redes elásticas [*elastic nets*], propuestas en 2005, combinan la regularización L1 con la regularización L2.

Al utilizar el gradiente descendente, la regla de aprendizaje resultante será de la forma

$$\Delta w_k = -\eta \frac{\partial L}{\partial w_k} = -\eta \left(\frac{\partial E}{\partial w_k} + \lambda_1 \text{sign}(w_k) + \lambda_2 w_k \right)$$

La regularización de redes elásticas se propuso para evitar los inconvenientes conocidos de la regularización L1:³⁹⁷

- Cuando el número de variables p es mayor que el número de observaciones n , LASSO (L1) selecciona como mucho n variables.
- Cuando el número de variables p es menor que el número de observaciones n y existe correlación entre las variables, en la práctica, la regularización L1 suele obtener peores resultados que la regularización L2 desde el punto de vista cuantitativo.
- Cuando hay grupos de variables correlacionadas, LASSO tiende a seleccionar sólo una de ellas en cada grupo de forma aleatoria y a ignorar el resto. Esta característica, obviamente, puede suponer un problema si pretendemos entrenar una red neuronal que sea especialmente robusta frente a variaciones en las entradas.

Las dos últimas características (especialmente la última) hacen que la regularización L2 sea mucho más habitual cuando hablamos de redes neuronales artificiales.

Los métodos anteriores (L1, L2 y *elastic nets*) son las técnicas de regularización de la función de coste más populares y utilizadas en la práctica, si bien no son, ni mucho menos, las únicas que se han propuesto.

Una forma alternativa de regularización es la denominada eliminación de pesos [*weight elimination*],³⁹⁸ en la que el término de regularización es de la forma $\sum(w_k^2/w_0^2)/(1+w_k^2/w_0^2)$, donde w_0 es un factor de normalización de los pesos. Este parámetro adicional, cuando $w_k > w_0$, hace que el término de la sumatoria sea cercano a uno, por lo que se limita a contar el número de pesos. Cuando $w_k < w_0$, el término de regularización es proporcional a w_k^2 , por lo que equivale a la regularización L2 o *weight decay*. En función de la selección que hagamos del parámetro w_0 , el entrenamiento de la red buscará soluciones con pocos pesos, aunque sean grandes (w_0 reducido), o muchos pesos pequeños (w_0 elevado).

Otras propuestas de regularización incorporan la matriz jacobiana (de derivadas parciales de cada salida con respecto a cada entrada) en el proceso de regularización, como es el caso del suavizado de pesos [*weight smoothing*], o permiten definir distintos tipos de regularizadores, como las redes de regularización generalizadas [*GRNs: generalized regularization networks*].

En el caso de las redes GRN,³⁹⁹ el término de regularización es un funcional (una función de una función) que se utiliza para suavizar la

³⁹⁷ Hui Zou y Trevor Hastie. Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society (Series B: Statistical Methodology)*, 67(2):301–320, 2005. DOI: 10.1111/j.1467-9868.2005.00503.x

³⁹⁸ Andreas S. Weigend, David E. Rumelhart, y Bernardo A. Huberman. Generalization by weight-elimination with application to forecasting. En *NIPS'1990 Advances in Neural Information Processing Systems 3*, pages 875–882. Morgan-Kaufmann, 1990. URL <https://goo.gl/864Mto>

³⁹⁹ Tomaso Poggio y Federico Girosi. Networks for approximation and learning. *Proceedings of the IEEE*, 78(9):1481–1497, Sep 1990. ISSN 0018-9219. DOI: 10.1109/5.58326

respuesta de la red, en este caso una función $\phi(f)$ de la operación f que realiza la red para aproximar el conjunto de datos de entrenamiento. Este funcional se puede derivar a partir de estabilizadores, los cuales pueden definirse para distintos tipos de redes, como las redes de base radial RBF, $G(r) = e^{-\beta r^2}$, como un producto de bases, $G(x) = \prod g(x_k)$ (p.ej. usando la transformada de Fourier), o como una suma de estabilizadores unidimensionales, $G(x) = \sum \theta_k g(x_k)$ (p.ej. con *splines*).⁴⁰⁰ Este modelo es muy general, aunque se planteó para redes con una única capa de neuronas ocultas, suficientes como aproximadores universales de funciones desde el punto de vista meramente matemático.

En el suavizado de perfiles jacobianos, el término de regularización es de la forma $(1/2P) \sum_p \sum_k \Omega(J_{*k}(W, x^p))$, donde x^p es uno de los P ejemplos de entrenamiento, J el jacobiano neuronal (la matriz de derivadas $\partial y_k / \partial x_i$) y Ω es un regularizador. Por ejemplo, puede utilizarse un regularizador de Tikhonov definido sobre el jacobiano neuronal $\Omega(J_{*k}) = (B \cdot J_{*k})^2$, siendo B una matriz de regularización. Si B es la matriz identidad, el regularizador minimizará la magnitud del jacobiano, igual que la regularización L2 minimiza la magnitud del vector de pesos. Obviamente, el coste computacional de este tipo de técnicas de regularización puede ser mucho mayor que el de las técnicas más comunes (L1, L2 y *elastic nets*). Para mejorar su eficiencia computacional, se propuso el suavizado de pesos [*weight smoothing*],⁴⁰¹ que aproxima el perfil jacobiano combinando la regularización L2 con un término adicional. El regularizador $(B \cdot J_{*k})^2$ se descompone hasta obtener un término de regularización que combina *weight decay* con regularizadores de preprocesamiento que suavizan las contribuciones de las entradas a las capas ocultas de la red: $(1/2P) \sum_p (\gamma \sum_k w_k^2 + \sum_j (B \cdot W_{*j})^2)$, siendo γ un parámetro de regularización adicional que determina la importancia relativa de la regularización L2 ($\sum_k w_k^2$) con respecto a los regularizadores $(B \cdot W_{*j})^2$.

Otras propuestas que se han efectuado están directamente orientadas a la obtención de representaciones dispersas de los vectores de datos de entrada, como es el caso de OMP [*Orthogonal Matching Pursuit*],⁴⁰² propuesto para el procesamiento digital de señales, como seguramente habrá podido imaginar por su nombre. El método OMP-k resuelve el problema de optimización con restricciones que obtiene el valor de los niveles de activación h que minimizan la función $\|x - Wh\|^2$ sujeta a la restricción $\|h\|_0 < k$, donde $\|h\|_0$ es la norma L0 (el número de entradas de h distintas de cero). Este problema se puede resolver de forma eficiente si obligamos a que la matriz W sea ortogonal, hecho del que proviene el nombre del método. Adam Coates y Andrew Ng evaluaron el uso de OMP, junto con otras técnicas de cuantificación vectorial y codificación dispersa, para extraer, de forma no supervisada, características de imágenes que luego se podían aprovechar como entradas en el entrenamiento de redes

⁴⁰⁰ Federico Girosi, Michael Jones, y Tomaso Poggio. Regularization theory and neural networks architectures. *Neural Computation*, 7(2):219–269, March 1995. ISSN 0899-7667. DOI: 10.1162/neco.1995.7.2.219

⁴⁰¹ Filipe Aires, Michel Schmitt, Alain Chedin, , y Noelle Scott. The weight smoothing regularization of MLP for Jacobian stabilization. *IEEE Transactions on Neural Networks*, 10(6):1502–1510, Nov 1999. ISSN 1045-9227. DOI: 10.1109/72.809096

⁴⁰² Y. C. Pati, R. Rezaiifar, y P. S. Krishnaprasad. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. En *Proceedings of 27th Asilomar Conference on Signals, Systems and Computers*, pages 40–44 vol.1, Nov 1993. DOI: 10.1109/ACSSC.1993.342465

neuronales para clasificar imágenes.⁴⁰³

En definitiva, las técnicas de regularización de la función de coste promueven que algunos de los pesos sinápticos de la red tengan valores cercanos a cero. De esta forma, se regula la capacidad de la red, haciéndola menos propensa al sobreaprendizaje y, de paso, se disminuye la probabilidad de que algunos pesos tomen valores desorbitados, fuera de control, durante el entrenamiento de la red, lo que también puede contribuir a disminuir ligeramente el error de la red sobre el conjunto de entrenamiento.⁴⁰⁴

En cuanto a la implementación de las técnicas de regularización de la función de coste, lo usual es regularizar los pesos de la red pero no los sesgos de las neuronas. ¿Por qué? Porque el sesgo de una neurona no interacciona de forma multiplicativa con sus entradas y, por tanto, no controla la influencia de una entrada particular frente a las demás. Cada peso sináptico asociado a una entrada real de una neurona, sin embargo, especifica cómo interactúan dos variables, la entrada asociada al peso y la salida de la neurona. El sesgo, en cambio, controla sólo una variable: la salida de la neurona. Por este motivo, no se introduce demasiada varianza en el modelo dejando a los sesgos sin regularizar. Además, si decidísemos regularizar los sesgos de las neuronas, estaríamos introduciendo un sesgo innecesario en el modelo. Estaríamos impidiendo artificialmente que la red aprovechase los sesgos para ajustarse adecuadamente a los datos de entrenamiento. Este sesgo inducido ocasionaría *underfitting*, el efecto contrario al *overfitting* o sobreaprendizaje, y degradaría el rendimiento de la red en la práctica. Así pues, en la práctica, se regularizan los pesos asociados a las entradas variables de las neuronas pero no sus sesgos (que podemos ver como pesos asociados a una entrada fija igual a 1).

En contadas ocasiones, puede resultar deseable utilizar una penalización diferente en la regularización de los pesos para las diferentes capas de una red neuronal multicapa. Esto equivale, en el fondo, a añadir nuevos hiperparámetros al algoritmo de entrenamiento de la red, ya que se utilizará un parámetro de regularización λ diferente para cada capa de la misma. No es algo demasiado habitual y rara vez se usa salvo, tal vez, para la capa de salida de la red.

En cuanto a la determinación del valor adecuado para el parámetro de regularización λ , lo normal es comenzar inicialmente sin regularización ($\lambda = 0$) y buscar un valor adecuado para la tasa de aprendizaje η . Una vez elegido el valor de la tasa de aprendizaje η , se recurre al conjunto de validación para encontrar un buen valor para λ . Por ejemplo, podemos comenzar con $\lambda = 1$ e ir cambiando su orden de magnitud (multiplicando o dividiendo por 10) para ir mejorando el rendimiento de la red sobre el conjunto de validación. Una vez encontrado el orden de magnitud adecuado para el parámetro de regularización λ , podemos intentar afinar algo más. Tras establecer el valor definitivo de λ , es importante volver a

⁴⁰³ Adam Coates y Andrew Y. Ng. The importance of encoding versus training with sparse coding and vector quantization. En *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 921–928. Omnipress, 2011. ISBN 978-1-4503-0619-5. URL http://www.icml-2011.org/papers/485_icmlpaper.pdf

⁴⁰⁴ Don R. Hush y Bill G. Horne. Progress in supervised neural networks. *IEEE Signal Processing Magazine*, 10(1):8–39, Jan 1993. ISSN 1053-5888. DOI: 10.1109/79.180705

Los sesgos suelen requerir menos datos de entrenamiento para ajustarse adecuadamente, ya que sirven para modificar el umbral de activación de las neuronas de la red, mientras que encontrar los pesos adecuados requiere observar cómo interactúan entradas y salidas bajo diferentes condiciones.

reajustar el valor de la tasa de aprendizaje η .

Restricciones sobre los parámetros de la red

Una alternativa a la regularización de la función de coste utilizada para guiar el entrenamiento de la red consiste en realizar la regularización directamente sobre los valores de los pesos. Las formas más habituales de hacerlo consisten en imponer, de forma explícita, restricciones sobre los valores de los pesos, ya sea limitando su magnitud o vinculando los valores de diferentes pesos de la red (p.ej. compartiendo parámetros entre distintas neuronas).

El método de los multiplicadores de Lagrange, llamado así en honor al matemático Joseph Louis Lagrange, permite resolver problemas de optimización con restricciones. El método de Lagrange reduce un problema de optimización de n variables y m restricciones a un problema de optimización sin restricciones de $n + m$ variables.

¿Cómo? Imaginemos que nuestro problema original es la optimización de los parámetros de una red neuronal con n pesos. Esto es, queremos minimizar la función de error asociada a los pesos, $E(w)$. Además, queremos imponer una restricción sobre la solución que limite la norma euclídea del vector de pesos: $\|w\|_2 \leq k$. Esta restricción la normalizamos definiendo una función $\Omega(w) = \|w\|_2 - k \leq 0$.

Imaginemos que dibujamos las curvas de nivel o mapas de contorno tanto de la función de error $E(w)$ como de la restricción $\Omega(w)$, a modo de mapa del tiempo con isobaras. Estas curvas se intersectarán en múltiples puntos. Fijando un valor para la restricción $\Omega(w)$, p.ej. $\Omega(w) = 0$, podemos movernos a lo largo de la curva y ver cómo varía el valor de $E(w)$. Cuando la curva $\Omega(w) = 0$ toca tangencialmente, sin cortar, una curva de nivel de $E(w)$, estamos en un óptimo local de $E(w)$ restringido a los puntos de la curva $\Omega(w) = 0$. En ese punto, las tangentes de $E(w)$ y $\Omega(w)$ son paralelas. Generalizando, podemos imponer esa condición de tangencialidad haciendo que los vectores asociados a los gradientes de $E(w)$ y $\Omega(w)$ sean paralelos, ya que los gradientes son perpendiculares a las líneas de contorno. Así pues, queremos puntos en los que

$$\nabla_w E(w) = \lambda \cdot \nabla_w \Omega(w)$$

donde hemos introducido una constante escalar $\lambda \neq 0$ porque los gradientes han de ser paralelos pero su magnitud no tiene por qué coincidir. Esta constante se denomina multiplicador de Lagrange.

Para incorporar la restricción anterior en una ecuación, podemos definir una función auxiliar denominada lagrangiano \mathcal{L} :

$$\mathcal{L}(w, \lambda) = E(w) - \lambda \cdot \Omega(w)$$

El método de los multiplicadores de Lagrange consiste en resolver el

problema de optimización sin restricciones sobre el lagrangiano \mathcal{L}

$$\nabla_{w,\lambda}\mathcal{L}(w, \lambda) = \nabla_{w,\lambda}[E(w) - \lambda \cdot \Omega(w)] = 0$$

Si nos fijamos, el problema de optimización con restricciones original, sobre n variables, se ha convertido en un problema de optimización sin restricciones con $n + 1$ variables. En este problema, la función que hay que optimizar tiene exactamente la misma forma que una función de coste regularizada, en la que, al término asociado a la función de error que estuviésemos utilizando, L_E , le añadíamos un término adicional de regularización, L_R :

$$L = L_E + \lambda L_R$$

En nuestro caso, la equivalencia entre la función de coste regularizada y el problema de optimización con restricciones resuelto usando el método de los multiplicadores de Lagrange puede verse si hacemos $L_E = E(w)$ y $L_R = \Omega(w)$ tras cambiar el signo del parámetro λ .

En otras palabras, podemos conseguir el mismo efecto si regularizamos la función de coste o si imponemos directamente restricciones sobre los valores de los parámetros durante el entrenamiento de la red.

Si utilizamos regularización L2 (*weight decay*), cuanto mayor sea el parámetro de regularización λ , más estamos restringiendo los valores de los pesos. Sin embargo, no sabemos cuál es el valor adecuado de λ para que los pesos sean de la magnitud que queramos, ya que la relación entre λ y el vector de pesos w dependerá de la función de error $E(w)$. La derivación anterior, no obstante, nos permite sustituir la penalización que realizábamos sobre la función de coste por una restricción explícita sobre el vector de pesos, que tendrá efectos similares sin que tengamos que determinar el valor más adecuado del parámetro de regularización λ .

¿Cómo podemos imponer una restricción explícita sobre el vector de pesos? Simplemente, ejecutamos el algoritmo de entrenamiento basado en el gradiente descendente sobre la función de error $E(w)$ y, tras cada actualización de pesos, si no se verifica la restricción $\|w\|_2 \leq k$, reproyectamos el vector de pesos w para que verifique la restricción. Esta transformación resulta útil si tenemos una idea de cuál debería ser el valor adecuado de k pero no queremos dedicar demasiado esfuerzo al ajuste del hiperparámetro de regularización λ .

Como efectos colaterales, el algoritmo de entrenamiento es menos probable que se quede atascado en óptimos locales correspondientes a valores pequeños de w (forzados por la regularización de la función de coste), que causan la aparición de unidades muertas [*dead units*], neuronas con pesos muy pequeños que no contribuyen demasiado a la salida de la red. La imposición de restricciones explícitas sólo tendrá efecto cuando los pesos crezcan demasiado y abandonen la región definida por la restricción $\|w\|_2 \leq k$. Un segundo efecto colateral de la imposición explícita de

restricciones será, entonces, estabilizar el algoritmo de optimización. Sin la restricción, o bien llega un momento en el que las neuronas se saturan y prácticamente se detiene el aprendizaje (desaparición del gradiente) o bien los pesos y los gradientes del error crecen cada vez más (explosión del gradiente), en un proceso de realimentación positiva que termina desencadenando problemas numéricos de desbordamiento (*overflow*).

Si se establece una cota superior sobre la magnitud del vector de pesos de cada neurona (su norma L2), se evita el uso de la regularización L2 y se habilita el uso de una tasa inicial de aprendizaje más alta, para explorar más rápidamente el espacio de posibles soluciones manteniendo la estabilidad del algoritmo.⁴⁰⁵ Cuando una actualización de pesos viola la restricción, los pesos se renormalizan, para lo cual sólo hay que realizar una simple división, p.ej. $w/\|w\|$ da como resultado un vector unitario en la misma dirección que el vector original w . Esta restricción se aplica, por separado, a las columnas individuales de la matriz de pesos de cada capa de la red neuronal, en vez de aplicarse a la matriz de pesos en su conjunto.⁴⁰⁶ De esta forma, ninguna neurona llegará nunca a tener pesos demasiado elevados y acaparar un protagonismo excesivo en el funcionamiento de la red neuronal. Además, las actualizaciones de los pesos siempre estarán acotadas, lo que facilita la estabilidad del algoritmo de entrenamiento de la red.

En la práctica, la imposición de una cota superior a la magnitud del vector de pesos de cada neurona se conoce como gradiente descendente proyectado (por lo de la proyección del vector de pesos cuando no se verifica la restricción). Lo más habitual es establecer una restricción explícita $\|w\|_2 < k$ que sustituya a la regularización L2, utilizando para k un valor que puede ser del orden de 3 ó 4. Formalmente, equivale a utilizar regularización L2 con un factor de regularización ajustado automáticamente para cada neurona por separado.

Aunque establecer restricciones sobre la norma del vector de parámetros es una forma efectiva de regularizar sus valores, hay una forma más popular de imponer restricciones sobre los valores de los pesos de una red: forzar que sean iguales. Al vincular los valores de unos pesos con los de otros, se reduce la capacidad efectiva de la red, con lo que se reduce también la posibilidad de que sobreaprenda. De esta forma, se consigue “regularizar” la red actuando sobre los valores de sus pesos, algo que se puede conseguir de distintas maneras:

- *Compresión de parámetros*

Comprimir el número de parámetros de la red consiste en representar el vector de pesos $w \in \mathbb{R}^K$ con menos de K parámetros.

Podemos reducir el número de parámetros de una red multicapa si sustituimos una capa de n entradas y m salidas por dos capas: una capa lineal de n entradas y p salidas seguida de una capa, con la

⁴⁰⁵ Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, y Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv e-prints*, arXiv:1207.0580, 2012b. URL <http://arxiv.org/abs/1207.0580>

⁴⁰⁶ Nathan Srebro y Adi Shraibman. Rank, Trace-Norm and Max-Norm. En Peter Auer y Ron Meir, editores, *COLT 2005: Proceedings of the 18th Annual Conference on Learning Theory, COLT 2005, Bertinoro, Italy, June 27-30, 2005.*, pages 545–560, 2005. ISBN 978-3-540-31892-7. DOI: 10.1007/11503415_37

misma función de activación que la capa original, pero de p entradas y m salidas. Las dos capas tienen un total de $np + pm = (n + m)p$ parámetros, número que puede ser significativamente menor que los nm parámetros de la capa original si elegimos adecuadamente el valor de p . Una red modificada de esta forma tiene menos parámetros que la red original, lo que puede facilitar su entrenamiento.

Podemos dar un paso más e intentar comprimir directamente la matriz de pesos de una capa. La idea consiste en descomponer la matriz de pesos W , de tamaño $n \times m$ como un producto $W = \Phi\alpha$, donde α será una matriz de parámetros de tamaño $p \times m$, con $p < n$, y Φ una segunda matriz de tamaño $n \times p$ que, a diferencia de la descomposición anterior en dos capas, no estará formada por parámetros ajustables, sino que tendrá valores fijos. Esta matriz puede elegirse, como en compresión de datos, a partir un conjunto de bases ortogonales o, incluso, muestreando una distribución aleatoria.⁴⁰⁷ Como resultado, un peso particular w_{ji} se generará a partir del espacio de parámetros comprimidos como $w_{ij} = \sum_k \phi_{ik}\alpha_{kj}$.

Comprimir la matriz de pesos de una capa de la red hace que la capa pase de tener $n \times m$ parámetros a sólo $n \times p$. Esta reducción es equivalente a comprimir la entrada de la capa utilizando una descomposición en dos capas en la que la primera capa de la red tiene conexiones sinápticas preestablecidas, que no se entrenan. Esto es fácil de ver, dado que $W^\top x = (\Phi\alpha)^\top x = (\alpha^\top\Phi^\top)x = \alpha^\top(\Phi^\top x) = \alpha^\top x'$, siendo x' la entrada que se obtiene comprimiendo la entrada original x con la transformación Φ .

Por tanto, la compresión de parámetros puede interpretarse como una etapa previa de preprocessamiento, que se diseña para reducir el coste computacional del proceso de entrenamiento (tanto el coste de cada iteración, como el número de iteraciones necesarias) por medio de la reducción del número de parámetros ajustables de la red. Como efecto secundario, también puede servir para prevenir el sobreaprendizaje.

■ Parámetros compartidos [parameter sharing]

Otra forma de reducir el número efectivo de parámetros de la red es compartirlos, ya sea entre distintos modelos (como en el aprendizaje multitarea y en el entrenamiento discriminativo) o entre distintos elementos dentro de una misma red (como en las redes convolutivas). Como, de esta forma, distintos modelos, o bien distintos componentes de un único modelo, comparten un conjunto único de parámetros, este método de regularización recibe el nombre de compartición de pesos o parámetros [*weight sharing* o *parameter sharing*].

El aprendizaje multitarea⁴⁰⁸ consiste en mejorar la capacidad de generalización de una red combinando los ejemplos disponibles para

⁴⁰⁷ Alexander Fabisch, Yohannes Kas-sahun, Hendrik Wöhrle, y Frank Kirchner. Learning in compressed space. *Neural Networks*, 42:83–93, 2013. ISSN 0893-6080. DOI: 10.1016/j.neunet.2013.01.020

⁴⁰⁸ Richard A. Caruana. Multitask connectionist learning. En *In Proceedings of the 1993 Connectionist Models Summer School*, pages 372–379, 1993

el aprendizaje de diferentes tareas en un mismo dominio. Esto es, se construyen redes neuronales especializadas para cada tarea, pero sus capas iniciales se comparten entre todas ellas. Los ejemplos de entrenamiento asociados a las diferentes tareas, implícitamente, están imponiendo restricciones débiles sobre los parámetros de las capas compartidas entre los distintos modelos. Dado que las primeras capas de una red, básicamente, se encargan de extraer características que luego se aprovechan en capas posteriores, combinar los ejemplos disponibles para varias tareas afines ayuda a extraer mejores características. El uso de estos parámetros compartidos es lo que permite que cada red individual sea, posteriormente, capaz de generalizar mejor.⁴⁰⁹

El entrenamiento discriminativo consiste en utilizar una estrategia semisupervisada que combine aprendizaje no supervisado con aprendizaje supervisado para regularizar los parámetros de un modelo supervisado.⁴¹⁰ De forma no supervisada, podemos construir un modelo “generativo” que describe la distribución observada de los datos de entrada, para lo que no necesitamos datos etiquetados y podemos disponer fácilmente de conjuntos de datos mucho más grandes. A continuación, podemos entrenar un modelo “discriminativo” utilizando técnicas de aprendizaje supervisado y regularizando sus parámetros para que sean similares a los parámetros del modelo no supervisado. Este entrenamiento discriminativo puede resultar útil cuando los datos etiquetados son escasos o resulta costosa su adquisición.

A partir del conocimiento específico que tengamos acerca del dominio concreto de aplicación donde se utilizará nuestra red neuronal, también podemos modelar dependencias entre los parámetros de una red. El mejor ejemplo de ello son las redes convolutivas que se utilizan para procesar imágenes. En una red convolutiva, la propia topología de la red ya codifica propiedades de las imágenes. Por ejemplo, si utilizamos un filtro de tamaño 3×3 , estamos indicando que los valores de píxeles vecinos contienen información útil. Además, si utilizamos el mismo filtro en diferentes posiciones de la imagen, usando siempre los mismos parámetros, estamos dotando a la red de invarianza frente a traslaciones: seremos capaces de identificar la misma característica local de una imagen en diferentes posiciones. Es algo habitual en visión artificial, no sólo en redes neuronales, ya que nos suele interesar que el procesamiento de las imágenes captadas por el sensor de una cámara sea invariante frente a los desplazamientos (traslaciones) de los objetos que aparecen en las imágenes. El caso de las redes convolutivas, que utilizan múltiples filtros del mismo tipo para diferentes posiciones de la imagen de entrada, es un ejemplo destacado de cómo se puede incorporar conocimiento del dominio en la arquitectura de una red neuronal. En este caso, además, se reduce significativamente el número de parámetros de la red, ya que no hay que ajustar los parámetros

⁴⁰⁹ Jonathan Baxter. Learning internal representations. En *Proceedings of the Eighth Annual Conference on Computational Learning Theory*, COLT '95, pages 311–320, 1995. ISBN 0897917235. DOI: 10.1145/225298.225336

⁴¹⁰ Julia A. Lasserre, Christopher M. Bishop, y T. P. Minka. Principled hybrids of generative and discriminative models. En *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 1, pages 87–94, June 2006. DOI: 10.1109/CVPR.2006.227

de los filtros para cada posición diferente de la imagen, sino que se utilizan los mismos parámetros para un filtro que se aplica sobre toda la imagen.

A diferencia de la regularización directa de los valores de los parámetros (por medio de una restricción sobre su norma/magnitud), al compartir parámetros, se reduce el número de parámetros de la red, con lo que se reduce el uso de memoria necesario. Además, esto suele facilitar el entrenamiento de la red y contribuir a la prevención del sobreaprendizaje. Recordemos que las primeras redes neuronales profundas que se pudieron entrenar con éxito fueron, precisamente, las redes neuronales convolutivas, gracias a las ventajas que ofrece su arquitectura especializada con parámetros compartidos.

Igual que pudimos establecer una correspondencia entre el uso de una función de coste regularizada (con regularización L2 o *weight decay*) y la imposición de restricciones sobre los parámetros de la red (limitando la norma del vector de pesos), también se puede construir una función de coste regularizada que corresponda, en cierto modo, a la imposición de vínculos entre los valores de los diferentes pesos de una red, como la que se realiza cuando se comparten parámetros [*weight sharing*]. La técnica, conocida como *soft weight tying*⁴¹¹ o *soft weight sharing*,⁴¹² añade un término de regularización a la función de coste que fomenta que los valores de los pesos se agrupen. Este agrupamiento se realiza de forma que los pesos de un mismo grupo tengan aproximadamente el mismo valor. El término de regularización añadido, en este caso, no es tan sencillo como el usado en la regularización L2, sino que se obtiene como una mezcla de múltiples distribuciones gaussianas. Está basado en un modelo GMM [*Gaussian Mixture Model*], usado en ocasiones para resolver problemas de aprendizaje no supervisado. Un modelo GMM es un modelo de tipo probabilístico que asume que una población de individuos se divide en subpoblaciones de tipo gaussiano (en este caso, en torno a diferentes valores de los pesos).

Introducción de ruido

Regularizar la función de coste o pérdida o imponer restricciones sobre los valores de los parámetros de una red neuronal son sólo dos de las alternativas de las que disponemos para intentar prevenir los efectos negativos del sobreaprendizaje. Una tercera alternativa consiste en añadir ruido pseudoaleatorio durante el proceso de entrenamiento de la red.

El efecto del ruido añadido artificialmente es similar, en cierta medida, a ampliar el conjunto de datos de entrenamiento. A diferencia de la ampliación del conjunto de entrenamiento utilizada para hacer robusta la red frente a determinadas transformaciones de los datos de entrada

⁴¹¹ Steven J. Nowlan y Geoffrey E. Hinton. Adaptive Soft Weight Tying using Gaussian Mixtures. En J. E. Moody, S. J. Hanson, y R. P. Lippmann, editores, *NIPS'1991 Advances in Neural Information Processing Systems 4*, pages 993–1000. Morgan-Kaufmann, 1991. URL <https://goo.gl/4J9onY>

⁴¹² Steven J. Nowlan y Geoffrey E. Hinton. Simplifying Neural Networks by Soft Weight-sharing. *Neural Computation*, 4(4):473–493, July 1992. ISSN 0899-7667. doi: 10.1162/neco.1992.4.4.473

(p.ej. rotaciones, traslaciones y cambios de escala en imágenes) o frente a posibles ataques malintencionados (como en el entrenamiento con adversario), la adición de ruido contribuye, de forma genérica, a hacer más robusta la red entrenada.

¿Cómo se le puede añadir ruido al entrenamiento de una red neuronal? Una vez más, tenemos varias alternativas a nuestra disposición. Podemos, directamente, añadir ruido a las entradas, ampliando el conjunto de entrenamiento para que la red sea robusta frente al tipo de ruido que esperamos encontrarnos en la práctica. En vez de actuar sobre las entradas que recibe la capa de entrada de la red, también podemos actuar sobre los niveles de activación de las capas ocultas, algo que podríamos interpretar como añadir ruido a las entradas a diferentes niveles de abstracción, los correspondientes a las características de los datos de entrada que las neuronas de las capas ocultas son capaces de extraer. Una tercera opción consiste en añadir ruido a los pesos sinápticos de la red. Veamos, por encima, en qué consiste cada una de estas opciones.

Introducción de ruido sobre las entradas

La primera forma de añadir ruido al proceso de aprendizaje de los parámetros de una red neuronal consiste en ampliar el conjunto de datos de entrenamiento añadiendo nuevos casos de entrenamiento, que construimos artificialmente introduciendo una pequeña cantidad de ruido aleatorio en los patrones originales del conjunto de entrenamiento.

En muchos problemas, estaremos interesados en que una red neuronal sea capaz de procesar datos de entrada que le lleguen con ruido añadido. Ese ruido, que puede ser aditivo o multiplicativo en función del problema, puede distorsionar los datos de entrada y confundir a una red neuronal que se haya entrenado con un conjunto de datos cuidadosamente preprocesado para no contener anomalías ni artefactos extraños, más conocidos como ruido. La red, en esos casos, no será demasiado robusta frente a la presencia de ruido en la práctica. Así pues, si somos capaces de modelar el tipo de ruido con el que la red tendrá que enfrentarse en la práctica, resulta aconsejable ampliar el conjunto de datos de entrenamiento con patrones adicionales que incorporen ese tipo de ruido. De esta forma, se consigue mejorar la robustez de la red ante la presencia de ruido.

Al añadir ruido al conjunto de entrenamiento, además de permitir que la red sea capaz de tratar correctamente datos con ruido, también se mejora su capacidad de generalización sobre datos sin ruido. Este efecto puede atribuirse a que, al entrenarla con ruido añadido, se hace que más neuronas de la red contribuyan de forma efectiva en el funcionamiento de la red en su conjunto.⁴¹³ En cambio, sin aplicar ruido a las entradas de la red durante su entrenamiento, es más probable que bastantes neuronas acaben resultando irrelevantes [*dead units*]. Lo que, por un lado, nos

⁴¹³ Jocelyn Sietsma y Robert J.F. Dow. Creating artificial neural networks that generalize. *Neural Networks*, 4(1): 67–79, 1991. ISSN 0893-6080. DOI: 10.1016/0893-6080(91)90033-2

permite podar la red y reducir su tamaño, por otro lado la hace más sensible a posibles variaciones en los datos de entrada.

La introducción de ruido en los datos con los que entrenamos la red puede, además, facilitar la convergencia del algoritmo de entrenamiento, lo que puede hacerla recomendable incluso cuando no se espera que la red tenga que trabajar con datos con ruido (si es que tal situación existe en el mundo real).

Añadir ruido al conjunto de entrenamiento resulta estrictamente necesario en la práctica si la red debe funcionar sobre datos con ruido, algo muy habitual en multitud de aplicaciones, desde los sistemas de reconocimiento óptico de caracteres [*OCR: Optical Character Recognition*], en los que los documentos escaneados pueden contener distintos tipos de manchas, hasta los sistemas de reconocimiento de voz, en los que la señal de voz puede venir acompañada de distintos tipos de ruido.

En realidad, añadir ruido artificial a los datos de entrada no es más que un caso particular de la estrategia que ya vimos de ampliación del conjunto de entrenamiento. En cualquier sistema de procesamiento de imágenes, es recomendable simular las diferentes condiciones que esperamos encontrarnos cuando el sistema, una vez entrenado, comience a operar en el mundo real:

- En un sistema de reconocimiento de objetos en imágenes, se puede aumentar el conjunto de entrenamiento para que la red sea más robusta frente a traslaciones, rotaciones, cambios de escala o, incluso, de iluminación. Eso sí, siendo cuidadosos con el tipo de transformaciones que se aplican, ya que algunas podrían no tener sentido, por ejemplo, en un sistema de OCR, en el que tenemos que ser capaces de distinguir una ‘b’ de una ‘d’, las dos anteriores de una ‘p’, una ‘n’ de una ‘u’, una ‘z’ de una ‘s’, una ‘E’ de un ‘3’ o un ‘6’ de un ‘9’. Aparte de las transformaciones más comunes, añadir ruido artificial puede contribuir a mejorar la robustez de la red neuronal.⁴¹⁴
- En sistemas de reconocimiento de voz, la adición de ruido puede realizarse en forma de ruido blanco de fondo (si bien ese tipo de ruido puede eliminarse con facilidad en las etapas de preprocessamiento de la señal de sonido) o como ruido sobre un modelo acústico del tracto vocal del hablante [*VTLN: Vocal Tract Length Perturbation*].⁴¹⁵ Añadir este tipo de perturbaciones sobre los ejemplos del conjunto de entrenamiento permite dotar a un sistema de reconocimiento de voz de mayor robustez frente a variaciones habituales en la voz, ya sea de distintas personas o de una misma persona bajo diferentes condiciones, con lo que se mejora el rendimiento de los sistemas de reconocimiento automático del habla [*ASR: Automatic Speech Recognition*].⁴¹⁶

La inyección de ruido en las entradas también se utiliza en algunos algoritmos de aprendizaje no supervisado, como los autocodificadores

A menudo, algunas diferencias seremos incapaces de resolverlas a nivel de caracteres individuales, por lo que serán responsabilidad de módulos posteriores del OCR, necesarios para diferenciar, en función del contexto, una ele minúscula (‘l’) de un uno (‘1’) o de una i mayúscula (‘I’), así como una o mayúscula (‘O’) de un cero (‘0’).

⁴¹⁴ Yichuan Tang y Chris Eliasmith. Deep networks for robust visual recognition. En *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML’10, pages 1055–1062, USA, 2010. Omnipress. ISBN 978-1-60558-907-7. URL <http://icml2010.haifa.il1.ibm.com/papers/370.pdf>

⁴¹⁵ Navdeep Jaitly y Geoffrey E. Hinton. Vocal Tract Length Perturbation (VTLN) Improves Speech Recognition. En *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013. URL <https://goo.gl/mMEiHm>

⁴¹⁶ Xiaodong Cui, Vaibhava Goel, y Brian Kingsbury. Data augmentation for deep neural network acoustic modeling. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 23(9):1469–1477, September 2015. ISSN 2329-9290. DOI: 10.1109/TASLP.2015.2438544

de eliminación de ruido [*denoising autoencoders*].⁴¹⁷ Un autocodificador es una red multicapa con la que se pretende generar, como salida, la misma entrada que recibe. Un autocodificador con eliminación de ruido, como su propio nombre indica, puede servir para eliminar el ruido de su entrada, si bien su objetivo final no suele ser éste, sino ser capaz de representar adecuadamente los datos utilizando los niveles de activación de sus neuronas ocultas (una forma de reducción de la dimensionalidad si el número de neuronas de la capa oculta que utilizemos para representar los datos es menor que el número de entradas). En este caso, el aprendizaje de una buena representación se obtiene como efecto colateral de aprender a eliminar ruido en la entrada.

De la misma forma que fuimos capaces de establecer correspondencias entre algunas formas de regularizar la función de coste e imponer restricciones directamente sobre los parámetros de la red, también se pueden establecer algunas correspondencias entre la introducción de ruido en la entrada y otras formas de regularización. Por ejemplo, perturbar las entradas de la red utilizando un ruido gaussiano de tipo aditivo es completamente equivalente al uso de la regularización L2, más conocida como *weight decay*, cuando se trata de una red con unidades lineales y una función de error cuadrático, tipo SSE o MSE.⁴¹⁸ La equivalencia se pierde cuando nuestra red utiliza elementos no lineales (en la práctica, siempre) o utilizamos funciones de coste alternativas (a menudo), pero nos puede servir para razonar acerca de los efectos que tiene la introducción de ruido en la regularización de la red.

Introducción de ruido sobre los pesos

Aunque añadir ruido sobre las entradas de la red sea la estrategia más habitual, entre otros motivos porque nos permite seguir utilizando los mismos algoritmos de entrenamiento de la red, sin tocar una sola línea de código, el ruido se puede añadir sobre otros elementos de la red neuronal, como pueden ser sus parámetros.

La introducción de ruido sobre los pesos de la red [*weight noise* o *synaptic noise*] se ha empleado principalmente en el entrenamiento de redes recurrentes.⁴¹⁹ Como sucedía con el ruido sobre las entradas, añadir una cantidad controlada de ruido a los valores de los pesos puede contribuir a acelerar la convergencia del algoritmo de aprendizaje, además de tener un efecto regularizador (una mejora en la capacidad de generalización de la red). De las diferentes estrategias que se pueden utilizar para añadir ruido, la introducción de ruido aditivo en cada actualización de los pesos parece ser la más efectiva para prevenir la saturación prematura de las neuronas de la red y, de ese modo, agilizar su entrenamiento.

Añadir ruido gaussiano de media 0 sobre los pesos puede interpretarse desde un punto de vista bayesiano.⁴²⁰ Desde una perspectiva bayesiana,

⁴¹⁷ Pascal Vincent, Hugo Larochelle, Yoshua Bengio, y Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. En *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, 2008. ISBN 978-1-60558-205-4. DOI: 10.1145/1390156.1390294

⁴¹⁸ Christopher M. Bishop. Training with Noise is Equivalent to Tikhonov Regularization. *Neural Computation*, 7(1):108–116, January 1995a. ISSN 0899-7667. DOI: 10.1162/neco.1995.7.1.108

⁴¹⁹ Kam-Chuen Jim, C.L. Giles, y Bill G. Horne. An analysis of noise in recurrent neural networks: convergence and generalization. *IEEE Transactions on Neural Networks*, 7(6):1424–1438, Nov 1996. ISSN 1045-9227. DOI: 10.1109/72.548170

⁴²⁰ Alex Graves. Practical variational inference for neural networks. En J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *NIPS'2011 Advances in Neural Information Processing Systems 24*, pages 2348–2356. Curran Associates, Inc., 2011. URL <https://goo.gl/MUKhLs>

los valores correctos de los pesos de una red está sujetos a incertidumbre, por lo que se pueden representar mediante distribuciones de probabilidad. El uso de ruido en los pesos es una forma eficiente en la práctica de reflejar esa incertidumbre sin recurrir a técnicas más costosas computacionalmente para reflejar el comportamiento estocástico de una red neuronal.

Igual que sucedía con el ruido sobre las entradas, bajo determinadas circunstancias, la introducción de ruido sobre los pesos también puede interpretarse como una forma de regularizar la función de coste. La minimización de una función de coste J añadiendo ruido gaussiano en los pesos de una red multicapa es equivalente a añadir un término de regularización adicional proporcional a $\|\nabla_w y\|^2$. Este término de regularización promueve que los parámetros tomen valores en los que pequeñas perturbaciones de los pesos causen un efecto pequeño en la salida y de la red. De esta forma, se fomenta la estabilidad del modelo, ya que no sólo se buscan mínimos de la función de coste J , sino mínimos en regiones planas, en las que el modelo es poco sensible a pequeñas variaciones en sus pesos.⁴²¹

Introducción de ruido sobre las capas ocultas

El ruido también puede añadirse sobre los niveles de activación de las capas ocultas de la red. En este caso, se puede interpretar de forma similar al ruido sobre las entradas de la red, salvo que ahora el ruido actúa directamente sobre las características que se extraen de los datos de entrada, características representadas implícitamente por los niveles de activación de las neuronas ocultas de una red multicapa.

Los autocodificadores con ruido [*NAE, noisy autoencoders*] son una extensión de los autocodificadores con eliminación de ruido [*DAE, denoising autoencoders*] que muestran cómo se puede mejorar la capacidad de generalización de una red, en este caso para eliminar ruido, mediante la introducción de ruido adicional en los niveles internos de una red multicapa.⁴²²

Otra estrategia popular de regularización en *deep learning*, denominada *dropout*, puede interpretarse también como una forma de introducción de ruido (aunque nosotros la interpretaremos desde un punto de vista más intuitivo, como una forma de construir un ensemble completo a partir de una única red neuronal). En el caso de *dropout*, su funcionamiento puede verse como introducir ruido multiplicativo de Bernoulli sobre las neuronas de la red: o bien se utiliza una neurona (su actividad se multiplica por uno) o bien no se utiliza (su actividad se multiplica por cero).

La normalización por lotes [*batch normalization*], otra de las técnicas más populares en *deep learning*, se propuso como una forma de mejorar

⁴²¹ Sepp Hochreiter y Juergen Schmidhuber. Simplifying neural nets by discovering flat minima. En G. Tesaurro, D. S. Touretzky, y T. K. Leen, editores, *NIPS'1994 Advances in Neural Information Processing Systems 7*, páginas 529–536. MIT Press, 1995. URL <https://goo.gl/1tc7xy>

⁴²² Ben Poole, Jascha Sohl-Dickstein, y Surya Ganguli. Analyzing noise in autoencoders and deep networks. *arXiv e-prints*, arXiv:1406.1831, 2014. URL <http://arxiv.org/abs/1406.1831>

el ajuste de los parámetros de la red, reparametrizando el modelo para facilitar su entrenamiento. También puede verse como un proceso de introducción de ruido sobre las unidades ocultas de la red, tanto aditivo como multiplicativo, al estimar medias y varianzas a partir de los datos de un minilote. Ese ruido también puede tener un efecto regularizador sobre la red y, en ocasiones, puede llegar a hacer innecesario el uso de *dropout*.

El propio gradiente descendente estocástico podría llegar a verse como una forma de introducir ruido, esta vez en la estimación del gradiente que se utiliza para actualizar los parámetros de la red. De hecho, hay quien aboga por introducir ruido de forma explícita en el cálculo del gradiente.⁴²³

Independientemente de la estrategia particular que utilicemos para introducir ruido en el entrenamiento de una red neuronal, el ruido contribuye a conseguir que la función implementada por la red neuronal sea “suave” [*smooth*], en el sentido de que pequeñas fluctuaciones, en sus entradas o, tal vez, en sus parámetros, no harán que la salida de la red cambie demasiado. Esa es una de las suposiciones que se encuentra detrás de muchas técnicas de regularización, que pretenden controlar la capacidad de la red con la intención de mejorar su capacidad de generalización. No es de extrañar, por tanto, que se puedan establecer algunas correspondencias entre técnicas particulares de introducción de ruido y otros métodos más tradicionales de regularización, como la regularización de la función de coste.⁴²⁴

Early stopping

A parte de modificar la función de coste, introducir restricciones sobre los parámetros de la red o introducir, de algún modo, ruido en el proceso de entrenamiento, existe una técnica de regularización muy efectiva que se ha convertido en la más utilizada en *deep learning*. Además, es muy fácil de implementar, como mostramos a continuación. La técnica se podría denominar “parada temprana”, aunque todo el mundo la conoce por su denominación en inglés: *early stopping*. Se puede ver como una forma de “regularización en el tiempo” y los expertos recomiendan que se utilicen de forma casi universal.

En las actividades humanas, las tradiciones suelen desempeñar el papel de restricciones evolutivas. Una pequeña dosis de conservadurismo, un ligero sesgo a favor de la historia, puede protegernos de los bruscos cambios que algunas modas introducen. En ocasiones, las novedades que se popularizan en forma de modas se acaban convirtiendo en tradiciones, cuando demuestran su valor con el paso del tiempo. Sin embargo, en otras ocasiones no son más que desviaciones pasajeras de lo que podríamos considerar un comportamiento racional. A veces se trata de simples

⁴²³ Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, y James Martens. Adding gradient noise improves learning for very deep networks. *arXiv e-prints*, arXiv:1511.06807, 2015. URL <http://arxiv.org/abs/1511.06807>

⁴²⁴ Pravin Chandra y Yogesh Singh. Regularization and feedforward artificial neural network training with noise. En *Proceedings of the 2003 International Joint Conference on Neural Networks*, volume 3, pages 2366–2371 vol.3, July 2003. DOI: 10.1109/IJCNN.2003.1223782

oscilaciones debidas a intereses comerciales (como en las prendas de moda) o a resultados de estudios científicos (como cuando se encuentran propiedades beneficiosas para la salud en el huevo, el chocolate o el café, tras un período en el que se desaconsejaba su consumo por sus efectos perjudiciales). Otras veces no son más que modas pasajeras, que pueden resultar inofensivas, como colgar zapatos de cables [shoe tossing], incluso con intenciones artísticas [shoefiti]; potencialmente peligrosas, dependiendo de dónde se practiquen, como algunos *selfies* o perseguir *pokémons*; o claramente temerarias, como el *balconing*. En el entrenamiento de redes neuronales, el *early stopping* introduce el sesgo de la tradición en la cultura humana.

Habitualmente, suele resultar provechoso adoptar determinados cambios si parecen ir en la dirección adecuada, algo de lo que no siempre estaremos seguros. Cuando la información disponible es escasa, seguir nuestros primeros instintos puede ser mejor que realizar un análisis más detallado de la situación. Puede que la incertidumbre sea tan elevada, que la decisión más racional sea precisamente actuar de forma intuitiva. Eso es lo queharemos al utilizar *early stopping*.

El entrenamiento de una red neuronal es un proceso de tipo iterativo, en el que vamos ajustando los pesos de la red con la intención de que ésta sea capaz de resolver satisfactoriamente un problema. Sin embargo, en el proceso tradicional de entrenamiento de la red carecemos de señales externas, más allá del conjunto de entrenamiento, que nos indiquen cuándo estamos consiguiendo que la red generalice bien y cuándo está sobreaprendiendo. Por este motivo, utilizaremos un conjunto de datos de validación, independiente del conjunto de entrenamiento, que nos sirva de señal externa mediante la cual guiar el proceso de entrenamiento de la red o, para ser más precisos, determinar cuándo hemos de pararlo.

Del mismo modo que utilizamos un conjunto de validación para determinar la configuración óptima de los hiperparámetros del algoritmo de entrenamiento de la red, ahora utilizaremos un conjunto de validación para detectar cuándo la red ha dejado de aprender y comienza a sobreaprender. El conjunto de validación lo emplearemos para evaluar periódicamente el error de generalización de la red conforme avanza su entrenamiento. No utilizaremos el conjunto de entrenamiento porque nunca nos serviría como medida del error de generalización de la red (el error de resustitución, sobre el conjunto de entrenamiento, nunca será una buena estimación del error de generalización, debido a su sesgo).

En principio, si el algoritmo de entrenamiento de una red funciona adecuadamente, la medida de error sobre el conjunto de entrenamiento debería decrecer en cada iteración del algoritmo (tras cada muestra o minilote en el gradiente descendente estocástico o tras cada época en el aprendizaje por lotes). Esa señal nos sirve para confirmar que el algoritmo está convergiendo hacia un mínimo de la función de coste o pérdida J .

Obviamente, consumido en exceso, todo alimento puede llegar a ser perjudicial para la salud, hasta el agua.

En la vida real, cuanto más importante sea una decisión, más tiempo le dedicaremos normalmente a intentar tomarla racionalmente, aunque tampoco es cuestión de llegar a los extremos de Charles Darwin, <https://goo.gl/WNDPd7>.

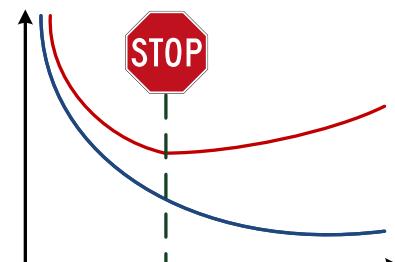


Figura 139: Al utilizar *early stopping*, el entrenamiento de la red se detiene en cuanto observamos que la función de coste deja de mejorar sobre el conjunto de validación (en rojo), aunque podamos seguir reduciéndola sobre el conjunto de entrenamiento (en azul).

Sin embargo, sobre el conjunto de validación, el error llega un momento en el que tiende a aumentar, cuando el modelo empieza a dar muestras de sobreaprendizaje. Cuando detectemos esa situación, deberíamos detener el algoritmo iterativo de aprendizaje y devolver los valores de los parámetros para los que el error sobre el conjunto de validación era mínimo.

El pseudocódigo del algoritmo de aprendizaje usando *early stopping* es el siguiente:

```
function (best, bestJ, bestN) = earlyStopping(net, training, validation)
    net.reset();
    best = net.parameters();
    bestJ = +Infinity;
    bestN = 0;
    n = 0
    attempts = 0;
    while (attempts<PATIENCE)
        n = n + STEPS;
        net.train(training, STEPS);
        currentJ = net.eval(validation)
        if (currentJ<bestJ)
            best = net.parameters();
            bestJ = currentJ;
            bestN = n;
            attempts = 0;
        else
            attempts++;
    
```

donde **net** es la red neuronal, **training** el conjunto de entrenamiento y **validation** el conjunto de validación. Además, hemos añadido dos parámetros más que aparecen en el pseudocódigo como constantes simbólicas: el número de actualizaciones de los pesos que realizamos entre cada par de evaluaciones sobre el conjunto de validación (**STEPS**) y nuestra paciencia (**PATIENCE**), el número de veces consecutivas tras las que, si no mejora la evaluación de J sobre el conjunto de validación, decidimos abandonar.

En cuanto a la paciencia que hemos de tener, podemos comenzar con una regla general de no detener el proceso hasta que llevemos 10 evaluaciones sin mejorar los resultados sobre el conjunto de validación. Posteriormente, conforme comprendamos mejor cómo se comporta la red particular que estamos entrenando, tal vez podamos permitirnos ser más indulgentes y permitir 20, 50 o más evaluaciones sin mejoras observadas sobre el conjunto de validación.

El entrenamiento de una red neuronal usando *early stopping* es un algoritmo eficiente para seleccionar el valor más adecuado de un hiperparámetro del algoritmo de aprendizaje: su número de pasos de entre-

namiento. De hecho, es tan eficiente que lo consigue hacer en una sola ejecución del algoritmo de entrenamiento. Se trata de una característica muy deseable que, por desgracia, no está disponible para la mayoría de los hiperparámetros que intervienen en el entrenamiento de una red neuronal.

Una vez establecido el número más indicado de pasos de entrenamiento (ya sean muestras en aprendizaje online, minibatches o épocas completas), si así lo deseamos, podemos entrenar la red con todos los datos del conjunto de entrenamiento completo (sin mantener apartado un conjunto de validación). Este aprendizaje final puede realizarse repitiendo el proceso completo, de `bestN` pasos, sobre el conjunto de datos completo, lo que generalmente nos proporcionará mejores resultados, aunque también será más costoso, al duplicar los recursos computacionales necesarios para entrenar la red neuronal.

Otra posibilidad consiste en continuar el proceso de entrenamiento de la red partiendo de los valores de los parámetros (`best`) que ya aprendimos al utilizar *early stopping*. El refinamiento de sus valores lo haremos ahora utilizando únicamente los datos del conjunto de validación. En este caso, nos surge una duda inmediata: ¿cuándo paramos? Un posible criterio es detener el proceso de ajuste final de los parámetros de la red cuando la función de coste J medida sobre el conjunto de validación quede por debajo de la que obtuvimos sobre el conjunto de entrenamiento al emplear *early stopping*; esto es, `bestJ`.

Entre las ventajas del uso de *early stopping* se encuentra el hecho de que reduce el coste computacional del entrenamiento de la red: paramos en cuanto el ajuste de los parámetros de la red comienza a dar señales de sobreaprendizaje. Además, *early stopping* proporciona un mecanismo de regularización bastante efectivo sin necesidad de que tengamos que manipular artificialmente la función de coste o introducir restricciones más o menos artificiales sobre los parámetros de la red. Obviamente, nada impide que podamos combinar *early stopping* con otras técnicas que tengan un efecto regularizador sobre la red, desde la introducción de ruido hasta la normalización por lotes [*batch normalization*] o el uso de *dropout*. Como cualquier otra técnica efectiva de regularización, puede contribuir a reducir la necesidad de preprocesar los datos de entrada.

Por otro lado, también se ha observado que el uso de *early stopping* tiene algunos efectos secundarios. Por ejemplo, tiende a restringir el espacio de búsqueda de valores adecuados para los parámetros de la red a una región, relativamente pequeña, en el entorno de los parámetros iniciales de la red. Además, el momento en el que se detiene el proceso de aprendizaje, decidido a partir del conjunto de validación, puede no ser el más adecuado para el conjunto total de datos disponibles para el entrenamiento de la red (el conjunto de entrenamiento usado para estimar sus parámetros y el de validación utilizado para tomar la decisión

de parar).⁴²⁵

Como sucedía con otras técnicas de regularización, también podemos establecer relaciones entre el uso de *early stopping* y la regularización de la función de coste: cuando se utiliza una función de error cuadrático, MSE o SSE, *early stopping* obtiene resultados similares a *weight decay*, la regularización L2.⁴²⁶ De hecho, si la red neuronal estuviese formada por elementos lineales, *early stopping* sería completamente equivalente a la regularización L2.

Un análisis formal más detallado muestra que los valores de los parámetros que corresponden a direcciones de mayor curvatura en la función de coste J se regularizan menos que los correspondientes a direcciones de menor curvatura. Esto se traduce en que los primeros se aprenden más rápido que los segundos.

La trayectoria que siguen los valores de los parámetros al utilizar regularización L2 termina en un mínimo de la función de coste regularizada. En el caso de *early stopping*, simplemente termina en un punto que se considere adecuado al monitorizar el error de validación. No obstante, la regularización L2 tiene el inconveniente de que requiere la realización de múltiples experimentos para ajustar el valor del parámetro de regularización, mientras que podríamos decir que *early stopping* determina de forma automática la cantidad adecuada de regularización del modelo.

En la práctica, por tanto, resulta muy habitual, casi omnipresente, el uso de *early stopping*. Es una forma sencilla, rápida y económica de regularizar un modelo y prevenir los efectos negativos del sobreaprendizaje.

Ensembles

Cuando comenzamos nuestro estudio sobre técnicas de prevención del sobreaprendizaje, comentamos que había tres formas de intentar evitar este fenómeno: conseguir más datos, ajustar la capacidad del modelo o emplear múltiples modelos. Las técnicas de regularización que hemos visto hasta ahora se centran en el segundo enfoque, ajustar la capacidad del modelo. La tercera estrategia a nuestra disposición consiste en combinar las predicciones de múltiples modelos, ya sea utilizando modelos diferentes [*model averaging*] o distintas instancias de un mismo tipo de modelo [*Bayesian fitting*].

El uso de varios modelos diferentes es una estrategia habitual en las técnicas de aprendizaje automático para mejorar la capacidad de generalización de cualquier modelo particular. Son los conocidos “ensembles”. Se basan en la idea de que modelos diferentes cometerán errores diferentes. Por este motivo, si combinamos múltiples modelos, podemos conseguir que, colectivamente, cometan menos errores que los modelos individuales de los que están compuestos.

La combinación de múltiples modelos se puede realizar de varias for-

⁴²⁵ J. Sjöberg y Lennart Ljung. Over-training, regularization and searching for a minimum, with application to neural networks. *International Journal of Control*, 62(6):1391–1407, 1995. DOI: 10.1080/00207179508921605

⁴²⁶ Christopher M. Bishop. Regularization and complexity control in feed-forward networks. En F. Fougelman-Soulie y P. Gallinari, editores, *Proceedings International Conference on Artificial Neural Networks ICANN'95*, volumen 1, páginas 141–148. EC2 et Cie, January 1995b. URL <https://goo.gl/ZKB7vj>

mas. Se pueden crear distintas muestras del conjunto de entrenamiento [*bootstrapping*] y construir distintos modelos a partir de esas muestras, que se combinan promediando sus predicciones, motivo por el que esta técnica se denomina *bagging*, un acrónimo prototípico formado por el fragmento inicial de *bootstrap* y el final de *aggregating*. Cada modelo individual se puede construir por separado, por lo que el proceso es fácilmente paralelizable. Algo que no sucede si empleamos *boosting*, donde los modelos se construyen iterativamente, con cada modelo nuevo intentando corregir los errores cometidos por los modelos anteriores. *Boosting* es más propenso al sobreaprendizaje, ya que su objetivo es aumentar la capacidad de los modelos individuales, no disminuir su varianza. Por este motivo, con redes neuronales, ya propensas al sobreaprendizaje, es mucho más frecuente el uso de *bagging*, que reduce la varianza de los modelos individuales promediando sus predicciones.

Un ensemble puede contener modelos de diferentes tipos entrenados a partir del mismo conjunto de entrenamiento. Es el caso de *stacking*, en el que un algoritmo de aprendizaje se utiliza para combinar las predicciones de otros algoritmos de aprendizaje, una forma de construir ensambles más general que el *bagging*. También existe la posibilidad de construir múltiples versiones de un mismo tipo de modelo, ya sea usando distintas muestras del conjunto de entrenamiento (como en *bagging*) o aprovechando la naturaleza estocástica de los algoritmos de aprendizaje. En el caso de las redes neuronales artificiales, las diferencias en la inicialización de los pesos, la selección aleatoria de minibatches y el uso de diferentes valores para los hiperparámetros del algoritmo de entrenamiento de la red pueden ser suficientes para introducir cierto grado de diversidad en los componentes de un ensemble, lo que permite mejorar la capacidad de generalización del ensemble con respecto a sus componentes individuales.

Cuando se utiliza una misma arquitectura de red neuronal, que se entrena en repetidas ocasiones para obtener diferentes configuraciones de la red con las que construir un ensemble (esto es, diferentes vectores de pesos), se suele hablar de ajuste bayesiano [*Bayesian fitting*]. Podemos implementar esta estrategia de diferentes formas:

- Se puede utilizar el mismo modelo, con el conjunto de hiperparámetros que resulte más adecuado para el problema particular, pero usando inicializaciones diferentes. No obstante, la diversidad de los modelos del ensemble puede resultar escasa, al deberse única y exclusivamente a las distintas inicializaciones.
- Se puede utilizar el mismo modelo, variando el conjunto de hiperparámetros. En el ensemble se incluyen aquellas variantes que mejores resultados hayan obtenido de todas las evaluadas. De esta forma, se introduce una mayor diversidad en el ensemble, aunque también se pueden incluir en él modelos subóptimos. Esta estrategia tiene la

Boosting se ha utilizado puntualmente, por ejemplo, para construir iterativamente una red a la que se le van añadiendo neuronas ocultas una a una.

Yoshua Bengio, Nicolas L. Roux, Paschal Vincent, Olivier Delalleau, y Patrice Marcotte. Convex neural networks. En Y. Weiss, P. B. Schölkopf, y J. C. Platt, editores, *NIPS'2005 Advances in Neural Information Processing Systems 18*, pages 123–130. MIT Press, 2006b. URL <https://goo.gl/DsBURU>

ventaja de que no requiere un proceso previo para determinar cuáles son los valores más adecuados de los hiperparámetros.

- Se puede, incluso, aprovechar el entrenamiento de un único modelo, del que nos quedamos con los valores de sus parámetros en diferentes momentos de su entrenamiento. Puede resultar útil cuando el entrenamiento del modelo es muy costoso computacionalmente, si bien los miembros del ensemble no serán independientes y los resultados tal vez no sean mucho mejores que los que podríamos obtener usando *early stopping* directamente.
- Una variante de la alternativa anterior es mantener una segunda red en la que sus parámetros sean una media móvil de los parámetros de la red que estemos entrenando. Esta versión “suavizada” de la red suele conseguir resultados ligeramente mejores que la red original. De forma intuitiva, si vemos el proceso de ajuste de parámetros como la construcción de una trayectoria sobre una superficie de error con forma de cuenco, en la que a veces nos quedamos cortos y a veces nos pasamos, la media es probable que esté en algún sitio cercano al mínimo de esa superficie de error.

En la práctica, el uso de múltiples modelos independientes suele ayudar a mejorar el rendimiento de un modelo de aprendizaje automático, ya se trate de una red neuronal o de cualquier otro tipo. Conforme aumenta el número de modelos en el ensemble, más aumenta el rendimiento del ensemble, si bien es cierto que con rendimientos decrecientes y coste proporcional al número de componentes del ensemble. Las mejoras proporcionadas por un ensemble suelen ser mayores cuanto mayor sea la independencia entre los diferentes modelos del ensemble. Usualmente, cuanta mayor diversidad, mejores resultados se obtienen. De ahí que, en muchas competiciones de Kaggle, los mejores resultados se obtengan utilizando ensembles, igual que sucedió en la competición del millón de dólares de Netflix.

Una desventaja del uso de ensembles es su consumo de recursos computacionales, proporcional al número de modelos del ensemble. Para reducir los requisitos computacionales de un ensemble, se puede intentar destilar el conocimiento de un ensemble en un único modelo.⁴²⁷ En un problema de clasificación supervisada, se entrena un modelo que imite las probabilidades de las distintas clases según el ensemble. Al aprender las probabilidades relativas de las distintas respuestas de acuerdo al ensemble, se consigue aprovechar su “conocimiento oscuro” [*dark knowledge*],⁴²⁸ donde reside la información aprendida por el ensemble pero no disponible directamente en el conjunto de entrenamiento. Un resultado similar se consigue cuando se emplea *dropout*, la técnica con la que cerraremos nuestro repaso a los métodos de regularización de redes neuronales.

⁴²⁷ Cristian Bucilă, Rich Caruana, y Alexandru Niculescu-Mizil. Model compression. En *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 535–541, 2006. ISBN 1-59593-339-5. DOI: 10.1145/1150402.1150464

⁴²⁸ Geoffrey Hinton, Oriol Vinyals, y Jeff Dean. Dark knowledge. *TTIC Distinguished Lecture Series, Toyota Technological Institute at Chicago*, 2014b. URL <https://youtu.be/EK61htlw8hY>

Dropout

Dropout es una forma muy sencilla y efectiva de regularización propuesta recientemente:⁴²⁹ durante el entrenamiento de la red, se mantiene una neurona activa con una probabilidad p , usualmente $p = 0.5$, y su salida se anula o descarta [*drop*] con probabilidad $1 - p$.

A diferencia de las técnicas de regularización de la función de coste, no se modifica la función de coste. Directamente, se actúa modificando la estructura de la propia red:

- Se eliminan de forma aleatoria (y temporal) la mitad de las neuronas ocultas de la red (cuando $p = 0.5$), dejando intactas las neuronas de las capas de entrada y de salida (o, como mínimo, las de salida, que siempre resultan necesarias).
- Se propaga la entrada x a través de la red modificada y se propaga hacia atrás el error, también a través de la red modificada, como haríamos habitualmente para entrenar una red neuronal.
- Se actualizan los pesos de la red, igual que se hace siempre a partir de la estimación del gradiente del error (obviamente, sólo la mitad de las neuronas se están utilizando actualmente, por lo que sólo se actualizarán los pesos de esa mitad de neuronas).
- Se vuelve a repetir el proceso, restaurando las neuronas eliminadas [*dropout neurons*] antes de volver a seleccionar aleatoriamente qué neuronas se anularán a continuación.

La red entrenada de esta manera aprenderá un conjunto de pesos en condiciones diferentes a las habituales: en cada iteración se han eliminado la mitad de las neuronas ocultas (o una fracción dada por la probabilidad p). Sin embargo, cuando utilicemos la red queremos usar todas sus neuronas, por lo que tendremos que compensar que el entrenamiento se hizo sólo con la mitad de las neuronas activas. Esta compensación se consigue dividiendo por dos los pesos correspondientes a las capas en las que se utilizó *dropout* (o, para un valor arbitrario del hiperparámetro p , multiplicando por p). Así conseguiremos que la entrada recibida por cada neurona de la red sea, durante su uso, similar a la que recibía durante su entrenamiento, cuando se anulaba artificialmente la actividad de la mitad de las neuronas de la red.

La idea de *dropout* parte de la premisa de que, descartando neuronas de la red durante su entrenamiento, se previene que diferentes neuronas se coadaptén demasiado.⁴³⁰ De esta forma, se reduce significativamente el sobreaprendizaje de la red.

En cierto modo, se puede decir que tiene una inspiración biológica. La reproducción sexual involucra el intercambio de genes entre dos

⁴²⁹ Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, y Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL <http://jmlr.org/papers/v15/srivastava14a.html>

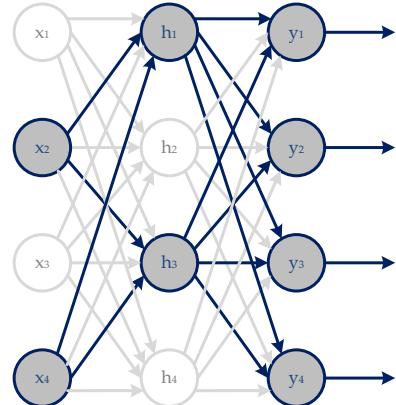


Figura 140: Entrenamiento con *dropout*, utilizando sólo una fracción de las neuronas internas de la red.

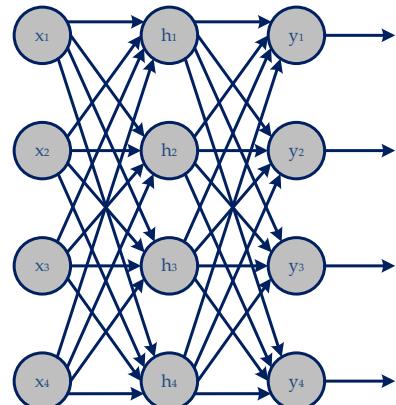


Figura 141: Uso de una red entrenada con *dropout*, considerando todas las neuronas de la red.

⁴³⁰ Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, y Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv e-prints*, arXiv:1207.0580, 2012b. URL <http://arxiv.org/abs/1207.0580>

organismos diferentes y crea una presión evolutiva para genes no sólo que sean buenos de por sí, sino que se puedan intercambiar fácilmente entre distintos organismos. Los genes son, entonces, más robustos frente a posibles cambios en su entorno, evitando su adaptación en exceso a las características extremadamente peculiares de un organismo particular. De la misma forma, las neuronas entrenadas con *dropout* tenderán a ser útiles por sí mismas, sin depender de las contribuciones de otras neuronas de su misma capa, en cuya presencia no pueden confiar⁴³¹ (recordemos que, en cada iteración del proceso de entrenamiento, se anula la mitad de las neuronas). Es una forma de conseguir que el aprendizaje de las neuronas ocultas no sea demasiado dependiente del contexto, sino que sea capaz de extraer características útiles bajo múltiples circunstancias, lo que hace a la red más robusta.

El uso de *dropout* se puede considerar equivalente a construir un ensemble, usando *bagging*, que está formado por un conjunto enorme de redes neuronales, pero sin necesitar el coste computacional que algo así conllevaría. Este ensemble estaría formado por todas las subredes que pueden formarse a partir de la red original eliminando unidades no de salida de la red de partida. Así pues, el ensemble incluiría un número exponencial de modelos individuales. La ventaja que proporciona *dropout* es que podemos construir algo más o menos equivalente a ese ensemble entrenando una única red neuronal.⁴³²

Imaginemos que entrenamos diferentes redes neuronales con la misma arquitectura a partir del mismo conjunto de datos de entrenamiento. Las redes, cuya configuración inicial de sus parámetros será diferente, terminarán siendo ligeramente distintas unas de otras. Aunque su diversidad tal vez no sea excesiva, será la suficiente como para poder construir un ensemble utilizando una votación ponderada [*bagging*]. Cuando, durante el entrenamiento con *dropout*, descartamos distintos subconjuntos de neuronas, el efecto es similar al que conseguiríamos entrenando distintas redes neuronales. Desde este punto de vista, *dropout* promedia los efectos de combinar un número muy grande de redes neuronales. Aunque las distintas redes, por separado, sobreaprendan de distintas formas, la combinación de sus resultados, ya sea con *bagging* o con *dropout*, reducirá la varianza del modelo obtenido y permitirá prevenir el sobreaprendizaje de la red.

Si utilizásemos *bagging*, no obstante, el coste computacional de esta estrategia la haría inviable en la práctica. Durante el uso del ensemble, denominado “inferencia” en este contexto, resultaría intratable tratar de considerar todas las redes de la familia para realizar cada predicción (tenemos un número exponencial de redes que se pueden derivar de suprimir neuronas ocultas de nuestra red original).

Como mucho, podríamos muestrear 10 ó 20 redes, usando diferentes máscaras para anular distintos subconjuntos de la red original, con las

⁴³¹ Alex Krizhevsky, Ilya Sutskever, y Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. En *NIPS’2012 Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL <https://goo.gl/Ddt8Jb>

⁴³² David Warde-Farley, Ian J. Goodfellow, Aaron Courville, y Yoshua Bengio. An empirical analysis of dropout in piecewise linear networks. En *ICLR’2014 International Conference on Learning Representations*, volume arXiv:1312.6197, 2014. URL <http://arxiv.org/abs/1312.6197>

que aproximar la predicción del ensemble completo, aunque sea de forma muy burda. Entonces tendríamos que promediar las salidas de la red, para lo cual utilizaremos la media geométrica en lugar de la media aritmética, que sería más habitual en otro tipo de ensambles pero que no correspondería a la operación que se realiza el ensemble construido con *dropout*.

Una forma más eficiente de realizar este muestreo consiste en utilizar todas las neuronas de la red y multiplicar sus salidas por la probabilidad p de que se incluya cada neurona en uno de los modelos del ensemble, para lo que basta un único recorrido de la red [*weight scaling inference rule*]. De esta forma, la salida utilizada captura el valor esperado de la neurona correspondiente en el ensemble completo (un promedio de un número exponencial de subredes posibles) y la entrada total de una neurona de la red es, durante su uso (inferencia), la misma que la que recibía durante su fase de entrenamiento. No existe una razón teórica que justifique esta aproximación para redes no lineales en general, sólo para casos particulares, pero en la práctica funciona muy bien.

Una forma alternativa de conseguir el mismo efecto es, una vez entrenada la red con *dropout*, multiplicar los pesos por p (dividirlos por dos si usamos $p = 0.5$) y utilizar la red resultante como cualquier otra red neuronal. Es decir, para su uso, hacemos que $W_{uso} = p W_{entrenamiento}$ en las capas afectadas por el uso de *dropout* durante el entrenamiento de la red.

Para implementar *dropout* de forma modular, podemos crear un nuevo tipo de capa, que denominaremos **DropoutLayer** y que podremos intercalar entre las diferentes capas de una red neuronal. En una red neuronal multicapa, lo único que tenemos que hacer es anular selectivamente las salidas de las neuronas no de salida de una red. Si implementamos este filtro como una capa intermedia, con el mismo número de entradas que de salidas, el resultado será similar al siguiente:

```
class DropoutLayer(n): Layer(n,n)
    function y = forward(x)
        mask = (random.vector(n) < p);
        y = x .* mask;
    function backward(error)
        delta = error;
    function y = use(x)
        y = p * x;
```

La capa de *dropout* tiene un único hiperparámetro, p , que corresponde a la probabilidad de que una neurona se utilice en cada momento. Habitualmente, se utiliza $p=0.5$ para las neuronas ocultas de la red y un valor más elevado, p.ej. $p=0.8$, para las neuronas de la capa de entrada (si es que decidimos aplicar *dropout* también a las neuronas de entrada).

Por motivos evidentes, las neuronas de la capa de salida nunca se anulan.

Para cada actualización de los pesos, se muestrea aleatoriamente una máscara binaria, `mask`, que se aplica sobre las entradas de la capa de `dropout`. Esta máscara, a 1 con probabilidad p y a 0 con probabilidad $1-p$, se utiliza para anular selectivamente algunas salidas y conservar únicamente una fracción p de los niveles de activación de la capa. En el código anterior hemos supuesto que la salida, una vez anulada, será 0, si bien esto no será válido para todos los tipos de redes neuronales. Por ejemplo, en las redes RBF tendríamos que modificar este procedimiento, ya que las capas de una red RBF evalúan la similitud (o diferencia) entre su vector de entradas y un valor de referencia dado por su vector de parámetros.

Para propagar la señal de error hacia atrás no tenemos que hacer nada. Sin embargo, cuando lo que queramos sea utilizar la red, una vez finalizado su entrenamiento, hemos de acordarnos de emplear el método `predict`, que ajusta las salidas de la capa de acuerdo a la probabilidad p . Si omitiésemos este cambio de escala, las entradas netas que recibirían las neuronas de la red durante el uso no se corresponderían con las que recibieron durante su entrenamiento y la red no funcionaría correctamente. Si la salida de una neurona, antes de usar `dropout`, era x , tras incorporar `dropout` al entrenamiento de la red, su salida esperada será $px + (1 - p)0$, ya que anulamos su salida con probabilidad $1 - p$. Cuando usemos todas las neuronas de la red, tendremos que ajustar sus salidas a px , como muestra el fragmento de pseudocódigo de arriba.

Para no tener que multiplicar por p cada vez que usemos la red, lo que supone un desperdicio de recursos computacionales, podemos “invertir” el proceso de `dropout`, aplicando el escalado durante el entrenamiento de la red, lo que nos permite no tener que hacer nada cuando se utiliza la red:

```
class InvertedDropoutLayer(n): DropoutLayer(n)
    function y = forward(x)
        mask = (random.vector(n) < p) / p;
        y = x .* mask;
    function backward(error)
        delta = error;
    function y = use(x)
        y = x;
```

En este caso, la máscara `mask` deja de ser binaria, ya que la dividimos por p al entrenar la red. A cambio de una operación adicional durante cada paso del entrenamiento de la red, nos ahorramos tener que realizar una multiplicación durante el uso de la red en la práctica. Dado que el rendimiento de la red durante su uso suele ser crítico en muchas aplicaciones, la versión invertida de `dropout` suele ser recomendable.

Obviamente, el mismo efecto, aunque algo menos elegante en su implementación modular, lo podemos conseguir si, una vez entrenada la red con *dropout*, eliminamos por completo las capas de tipo **DropoutLayer**, para lo que tendremos que ajustar proporcionalmente los pesos de las capas que reciben la salida de las capas de *dropout*, multiplicándolos por p . El que usa la red, de esta forma, ni siquiera tiene por qué saber si se utilizó *dropout* durante el entrenamiento de la red.

Independientemente de la alternativa de implementación por la que nos decantemos, el resultado final será siempre el mismo. *Dropout* puede verse como una forma de muestrear una red neuronal de una familia enorme de posibles redes neuronales (todas las subredes con una fracción p de las neuronas ocultas de la red original) y actualizar los pesos de esa red muestreada utilizando los datos de entrenamiento. Durante su uso, ya no se aplica *dropout*. Ya no se descartan partes de la red, sino que se utiliza toda la red para obtener un resultado que puede interpretarse como una estimación de la predicción media realizada por un ensemble de tamaño exponencial. A diferencia de otras técnicas de construcción de ensembles, en este caso, las redes incluidas en el ensemble no son independientes, ya que comparten sus parámetros (otra forma de controlar la capacidad de un modelo).

El uso de *dropout* se puede considerar no sólo como equivalente a la construcción de un ensemble usando *bagging*, sino que también se puede conectar con otras de las técnicas de regularización que ya hemos visto:

- *Introducción de ruido multiplicativo sobre las neuronas ocultas*

Las máscaras utilizadas por *dropout* para anular una fracción de las salidas de las neuronas ocultas de una red (y, tal vez, una fracción menor de sus entradas) pueden verse como una forma de ruido multiplicativo aplicado sobre las capas internas de la red. Este ruido añade un componente estocástico al entrenamiento de la red.

En vez de introducir el ruido en la entrada, la introducción de ruido en la actividad interna de la red puede considerarse una forma inteligente de destruir información de forma selectiva (la correspondiente a la extracción de características de los datos de entrada). Al eliminar individualmente esas características, estamos obligando a que la red sea capaz de aprender formas alternativas de llegar a las mismas conclusiones, ya sea identificando la misma característica de forma redundante en varias unidades de la red o construyendo otras características diferentes que puedan ser útiles para resolver el problema.

Para conseguir que la red sea robusta, el ruido introducido debe ser multiplicativo. Si el ruido fuese aditivo, sin anular la actividad de las neuronas ocultas de la red, la red aprendería a amplificar los niveles de actividad de esas neuronas para que el ruido fuese insignificante en

comparación con él, pero sin lograr el efecto regularizador del *dropout*. El ruido multiplicativo evita que la red se adapte fácilmente a la presencia de ruido y nos permite obtener soluciones más robustas.

- *Regularización de la función de coste*

Dropout puede verse como una forma de asegurarnos de que un modelo es robusto frente a la pérdida parcial de información (las neuronas cuya salida se anula). En este sentido, es similar también a los métodos de regularización de la función de coste, que tienden a reducir los valores de los parámetros de la red. Esta reducción de los valores de los parámetros de la red la hace más robusta frente a pérdidas de conexiones individuales y, en el caso de la regularización L1, tiende a hacerla más dispersa anulando muchos de los pesos.

Si aplicamos *dropout* a un problema de regresión lineal, se puede demostrar que *dropout* es equivalente a utilizar una regularización L2, *weight decay*, en la que se utilizan diferentes parámetros de regularización para cada entrada en función de su varianza.⁴³³

En la práctica, es habitual utilizar regularización L2 o *weight decay* para entrenar redes neuronales artificiales, utilizando un parámetro de regularización global λ previamente ajustado usando un conjunto de validación o validación cruzada. Esta regularización global se puede combinar con *dropout* aplicado sobre todas las capas ocultas de la red (y, opcionalmente, la capa de entrada). En cuanto al hiperparámetro p , el valor por defecto $p = 0.5$ suele ser razonable, aunque también podemos ajustarlo sobre el conjunto de validación si lo consideramos oportuno.

En el artículo original en el que se propuso esta técnica de regularización, se realizaban algunos experimentos con conjuntos de datos estándar como MNIST, CIFAR-10, ImageNet, TIMIT o Reuters.⁴³⁴ En el caso de MNIST, el uso combinado de *dropout* en las capas ocultas ($p = 0.5$) con una versión de la regularización L2, conseguía una tasa de error del 1.3 %, frente al 1.6 % del que nadie había conseguido bajar usando una red multicapa sin ampliar el conjunto de entrenamiento, utilizar arquitecturas especializadas que aprovechen las propiedades espaciales de las imágenes (esto es, redes convolutivas) o someter a la red a un proceso previo de pre-entrenamiento no supervisado. Si, además, se añadía el uso *dropout* sobre los píxeles de las imágenes de entrada, con $p = 0.8$ para la capa de entrada, la tasa de error se reducía hasta el 1.1 %: una reducción de casi un tercio del error con respecto a los mejores resultados obtenidos hasta ese momento con una red neuronal multicapa convencional.

A parte de las mejoras de rendimiento que ofrece el uso de *dropout*, una de las principales ventajas de esta técnica de regularización es que puede utilizarse fácilmente sobre cualquier tipo de red neuronal artificial, mientras que otras técnicas de regularización imponen restricciones sobre el tipo de red o las situaciones en las que resultan aplicables.

⁴³³ Stefan Wager, Sida Wang, y Percy S. Liang. Dropout training as adaptive regularization. En C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, y K. Q. Weinberger, editores, *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, pages 351–359. Curran Associates Inc., 2013. URL <https://goo.gl/fmekb8>

⁴³⁴ Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, y Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv e-prints*, arXiv:1207.0580, 2012b. URL <http://arxiv.org/abs/1207.0580>

Su popularidad ha ocasionado que se propongan diferentes variaciones que giran en torno a la misma idea. Al fin y al cabo, *dropout* no es más que una forma de aproximar la contribución de todos los submodelos incluidos en un ensemble de tamaño exponencial. Comentemos los resultados que se han obtenido con algunas de ellas:

- *Fast dropout*⁴³⁵

La versión original de *dropout* nos obliga a muestrear repetidamente un subconjunto de las neuronas de la red. Realizando una aproximación gaussiana de este proceso de muestreo, se puede conseguir un algoritmo de entrenamiento que converge más rápidamente, motivo por el que esta variante se denomina *fast dropout*. En cambio, el coste computacional del uso de la red será más elevado, ya que no bastará simplemente con un cambio de escala multiplicando por p , sino que habrá que repetir el mismo proceso de muestreo que se realiza durante el entrenamiento.

- *Dropout boosting*⁴³⁶

De la misma forma que *dropout* se puede considerar análogo a la construcción de un ensemble con *bagging*, *dropout boosting* consiste en entrenar la red utilizando una estrategia similar a *boosting*. El *dropout* convencional intenta maximizar la verosimilitud de la salida correspondiente a un ejemplo dada la subred actual (la muestra sobre la que trabaja). En *dropout boosting*, se tienen en cuenta las contribuciones de otras subredes, como en *boosting*. En el *boosting* tradicional, los demás modelos se han entrenado previamente y el modelo actual intenta corregir sus errores. Al usar *dropout boosting*, todos los modelos comparten sus parámetros pero, al principio, ninguno se ha entrenado, por lo que hay que modificar la regla de aprendizaje de *boosting*. Una posible solución consiste en realizar una estimación sesgada del gradiente del ensemble. Sin embargo, como cabría esperar, dado que *boosting* es más propenso al sobreaprendizaje que *bagging*, *dropout boosting* apenas consigue mejoras con respecto al gradiente descendente estocástico. Esto es, el ensemble resultante no es mejor que una simple red y, desde luego, no proporciona los beneficios que sí se obtienen con *dropout*.

- *DropConnect*⁴³⁷

Otra variante de *dropout* consiste en aplicar *dropout* a las conexiones sinápticas de la red, en vez de a neuronas completas. El resultado, denominado *DropConnect*, emplea máscaras sobre las conexiones individuales (de tamaño $n \times m$ en el caso de una capa completamente conectada de n entradas y m salidas). En lugar de aplicar las máscaras sobre los niveles de activación de las salidas de las neuronas, se aplica sobre los pesos correspondientes a las entradas individuales de cada neurona. Para realizar inferencias (esto es, usar la red), se realizan

⁴³⁵ Sida Wang y Christopher Manning. Fast dropout training. En Sanjoy Dasgupta y David McAllester, editores, *Proceedings of the 30th International Conference on Machine Learning*, volume 28(2) de *Proceedings of Machine Learning Research*, páginas 118–126, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/wang13a.html>

⁴³⁶ David Warde-Farley, Ian J. Goodfellow, Aaron Courville, y Yoshua Bengio. An empirical analysis of dropout in piecewise linear networks. En *ICLR'2014 International Conference on Learning Representations*, volume arXiv:1312.6197, 2014. URL <http://arxiv.org/abs/1312.6197>

⁴³⁷ Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, y Rob Fergus. Regularization of Neural Networks using DropConnect. En Sanjoy Dasgupta y David McAllester, editores, *Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) de *Proceedings of Machine Learning Research*, páginas 1058–1066, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/wan13.html>

múltiples muestras de las entradas de cada neurona (del orden de 1000) y se promedian las activaciones de las neuronas individuales. Al menos, este proceso se puede paralelizar con ayuda de una GPU y, en ocasiones, permite obtener resultados competitivos, algo mejores que los obtenidos con *dropout*.

- *Ruido multiplicativo gaussiano*⁴³⁸

Los promotores originales de *dropout*, en un artículo posterior publicado en JMLR, proporcionan una versión alternativa de *dropout* en la que, en vez de aplicar un ruido multiplicativo de Bernouilli (multiplicar por 1 cuando se preserva la actividad de la neurona, por 0 cuando se anula), se aplica un ruido multiplicativo gaussiano de acuerdo a una distribución normal con media 0. Sorprendentemente, esta variación funciona tan bien como la versión original con ruido de Bernouilli, quizá incluso hasta mejor.

Multiplicar por ruido gaussiano de media y varianza unitarias equivale a sumar ruido gaussiano con media cero y desviación estándar igual al nivel de activación de la neurona: $h_k r$ con $r \sim \mathcal{N}(1, 1)$ es equivalente a $h_k + h_k r'$ con $r' \sim \mathcal{N}(0, 1)$.

Cuando optamos por un ruido gaussiano multiplicativo, se recomienda utilizar ruido de acuerdo a una distribución normal $\mathcal{N}(1, \sigma^2)$, en el que la varianza σ^2 se convierte en un hiperparámetro del algoritmo que sustituye a la probabilidad p del ruido de Bernouilli. Los autores recomiendan utilizar $\sigma = \sqrt{(1-p)/p}$, donde p es el valor usado habitualmente en *dropout*. Empleando ruido gaussiano, los resultados experimentales obtenidos permiten bajar del 1% sobre la base de datos MNIST ($0.95 \pm 0.04\%$ realizando 10 experimentos partiendo de semillas aleatorias diferentes).

Lo más interesante de esta variante de *dropout* es que el valor esperado de los niveles de activación de las neuronas permanece inalterado, por lo que no hace falta reescalar los pesos cuando se usa la red (ni, por supuesto, recurrir procesos más costosos de muestreo, como sucede en *DropConnect*). Sólo tenemos que usar una máscara de ruido gaussiano durante el proceso de entrenamiento de la red y el resto funciona como en cualquier otra red neuronal:

```
class GaussianDropoutLayer(n): DropoutLayer(n)
    function y = forward(x)
        mask = random.normal(1,sigma).vector(n);
        y = x .* mask;
    function backward(error)
        delta = error;
    function y = use(x)
        y = x;
```

⁴³⁸ Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, y Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL <http://jmlr.org/papers/v15/srivastava14a.html>

A modo de resumen final de este capítulo dedicado al uso de técnicas de regularización para prevenir el sobreaprendizaje:

La idea básica de *dropout* consiste en descartar neuronas, junto con sus conexiones sinápticas, durante el proceso de entrenamiento de la red neuronal. Esto evita que las diferentes neuronas de la red se coadapten en exceso, lo que permite reducir significativamente el sobreaprendizaje y proporciona mejoras notables de rendimiento frente a otros métodos de regularización. Además de ser un excelente regularizador, *dropout* es fácil de implementar y compatible con muchos modelos de redes y algoritmos de entrenamiento.

La normalización por lotes, en ocasiones, también sirve para reducir el error de generalización de una red y permite que no tengamos que utilizar *dropout*, dado que las estimaciones que se realizan de los estadísticos (media y varianza) que se emplean para normalizar los niveles de activación en el procesamiento de cada minilote introducen ruido, tanto aditivo como multiplicativo, en el proceso de entrenamiento de la red.

Incluso aunque dispongamos de un conjunto de datos de entrenamiento grande, siempre se debería incluir algún tipo de regularización cuando se entrena una red neuronal. El uso de *early stopping* es casi obligado. La regularización L2, o *weight decay*, es muy habitual. *Dropout* es muy útil y también debería formar parte de nuestras herramientas habituales.

Si nos detenemos un poco, veremos que la gran variedad de métodos de regularización que hemos analizado se pueden interpretar desde un punto de vista intuitivo como estrategias genéricas que tratan de dotar a los modelos que construimos de propiedades potencialmente deseables.⁴³⁹ Esas propiedades pueden estar relacionadas con la suavidad [*smoothness*] del modelo, su linealidad (al menos, localmente), la extracción de jerarquías de características (aprendizaje de representaciones), la posibilidad de compartir esas características entre distintos modelos [*transfer learning*] o la capacidad de reducir la dimensionalidad de un problema y obtener modelos dispersos [*sparseness*].

Le hemos dedicado unas cuantas páginas a estudiar diferentes formas de regularización, que intentan guiar el proceso de aprendizaje para dotar a nuestros modelos de algunas de esas características que parecen ser deseables. Aunque, en muchas ocasiones, el objetivo se consigue de forma sutil y difícil de apreciar a simple vista, por suerte las técnicas de regularización son bastante fáciles de implementar. Pequeños cambios en el código que utilizamos para entrenar una red neuronal nos pueden permitir mejoras notables en el rendimiento de los modelos que se obtienen. No existe motivo alguno para que no nos aprovechamos de ello.

⁴³⁹ Yoshua Bengio, Aaron Courville, y Pascal Vincent. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, Aug 2013b. ISSN 0162-8828. DOI: 10.1109/TPAMI.2013.50

Algoritmos de optimización

Casi todos los algoritmos de deep learning involucran de alguna forma un proceso de optimización. En realidad, no se trata de una característica específica de las redes neuronales. Muchas técnicas de aprendizaje automático se basan en convertir el proceso de aprendizaje en un problema de optimización que podemos resolver utilizando herramientas de cálculo numérico. En el caso particular de las redes neuronales, ya sabemos cómo calcular el gradiente del error de forma eficiente utilizando backpropagation. En este capítulo, veremos cómo aprovechar la información que proporciona ese gradiente para ajustar los parámetros de una red neuronal artificial.

Para convertir un problema de aprendizaje en un problema de optimización, definimos una función de error, coste o pérdida $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$. Como pretendemos reducir ese error, que depende del conjunto x de parámetros de nuestro modelo, nuestro problema de aprendizaje se reduce al siguiente problema de minimización:

$$x^* = \arg \min f(x)$$

La función de pérdida $f(x)$ se convierte en la función objetivo de nuestro problema de minimización. En realidad, no importa si definimos el problema como un problema de minimización (del error del modelo) o de maximización (del “acerto” del modelo), ya que la minimización de la función $f(x)$ es equivalente a la maximización de la función $-f(x)$.

Las técnicas numéricas de optimización nos ayudarán a resolver nuestro problema de la forma más eficiente posible, intentando reducir los recursos computacionales necesarios, tanto en tiempo de CPU como en espacio en memoria. En el caso de las redes neuronales, el esfuerzo computacional viene determinado principalmente por el número de veces que hay que evaluar la función $f(x)$ (y su gradiente $\nabla f(x)$), por lo que intentaremos reducir el número necesario de evaluaciones de $f(x)$ (y $\nabla f(x)$).

Para entender por qué usamos el gradiente de la función de error, $\nabla f(x)$, viene bien recordar que el gradiente no es más que el vector de derivadas parciales de la función f con respecto a cada una de las variables sobre las que está definida la función. En el caso unidimensional, x es un valor real y el gradiente es la derivada $f'(x)$, la pendiente de la curva $f(x)$ en el punto x . En el entorno de x , se verifica lo siguiente:

$$f(x + \epsilon) \approx f(x) + \epsilon f'(x)$$

Si queremos encontrar el valor x^* que minimiza la función x , podemos aprovechar la información que nos proporciona la derivada (el gradiente en el caso multidimensional). Si la pendiente $f'(x)$ es positiva, deberíamos reducir el valor de x para acercarnos al mínimo de la función $f(x)$. Si la pendiente $f'(x)$ es negativa, deberíamos aumentar el valor de x . Es decir, la actualización de la variable x se debería realizar de acuerdo a la siguiente expresión:

$$\Delta x = -\eta f'(x)$$

donde η es un parámetro que determina el tamaño del salto (la tasa de aprendizaje).

Si generalizamos esta estrategia al caso multidimensional, en el que x es un vector de variables, obtenemos el método del gradiente descendente con el que ya estamos familiarizados y que fue propuesto por el matemático francés Augustin Louis Cauchy en 1847:⁴⁴⁰

$$\Delta x = -\eta \nabla f(x)$$

La expresión anterior la podemos interpretar como una combinación de dos factores: la dirección en la que se da un salto (determinada por el gradiente $\nabla f(x)$ de la función de error) y el tamaño del salto que damos (dados por la tasa de aprendizaje η). Se han propuesto numerosas técnicas numéricas de optimización que nos ayudan a resolver problemas de este tipo.⁴⁴¹ A grandes rasgos, las podemos clasificar en dos categorías:

- Técnicas de búsqueda lineal [*line search*]: Primero se elige la dirección en la que se da el salto (por ejemplo, la dada por el gradiente) y, en segundo lugar, se determina el tamaño óptimo del salto.
- Técnicas basadas en regiones de confianza [*trust region*]: En primer lugar se elige el tamaño del salto (el tamaño de la región de confianza) y, a continuación, la dirección en la que se da el salto.

Tanto unas como otras permiten garantizar la convergencia del algoritmo de optimización cuando la función optimizada es convexa: una secuencia de iteraciones de este tipo converge al óptimo global de la función. Sin embargo, la mayoría de las funciones que deseamos optimizar no son necesariamente convexas, por lo que los algoritmos iterativos de optimización alcanzarán un óptimo local de la función dependiendo de nuestro punto de partida inicial.

Volviendo a nuestro problema de aprendizaje automático, lo que nos gustaría conseguir es minimizar el error de generalización de nuestro modelo; esto es, su error evaluado sobre el conjunto de prueba, en ocasiones denominado “riesgo” [*risk*]. Sin embargo, durante el entrenamiento del modelo no disponemos de los datos sobre los que se usará el modelo en el mundo real, sino de los datos del conjunto de entrenamiento. Así

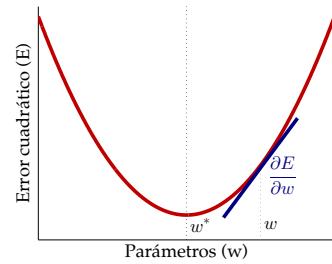


Figura 142: La función de error, coste o pérdida en el caso unidimensional: La derivada del error para el valor actual del peso w nos indica en qué sentido debemos corregir ese valor para acercarnos a su valor óptimo w^* , pero sólo si se trata de una función convexa, una aproximación usualmente válida en el entorno cercano del punto w .

⁴⁴⁰ Augustin-Louis Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Compte Rendu des Séances de L'Académie des Sciences XXV*, Série A(25):536–538, 1847

⁴⁴¹ William H. Press, Saul A. Teukolsky, William T. Vetterling, y Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007. ISBN 0521880688

pues, nuestro problema de optimización lo definimos sobre una función de error de nuestro modelo sobre el conjunto de entrenamiento, también conocido como error de resustitución o riesgo empírico [*empirical risk*]. Idealmente, el conjunto de entrenamiento será una muestra representativa de los datos reales sobre los que deberá trabajar nuestro modelo, por lo que nos servirá como aproximación válida. De esta forma, conseguimos que nuestro problema de aprendizaje se convierta en un problema de optimización. El entrenamiento de un modelo de aprendizaje automático se plantea como un problema de minimización del riesgo empírico.

Nunca debemos olvidar que, en aprendizaje automático, estamos optimizando una función (el riesgo empírico, evaluado sobre el conjunto de entrenamiento) distinta a la que realmente nos gustaría poder optimizar directamente (el error de generalización). Por este motivo, necesitamos utilizar técnicas de regularización que prevengan el sobreaprendizaje.

Técnicas como *early stopping* nos ayudarán a determinar el momento idóneo en el que detener nuestro proceso de optimización de los parámetros de una red neuronal artificial. En ocasiones, el entrenamiento de la red terminará antes de llegar a un óptimo local de la función de coste empleada: aunque podríamos seguir reduciendo el error sobre el conjunto de entrenamiento (el riesgo empírico), *early stopping* nos ayuda a decidir cuándo parar evaluando el error sobre un conjunto de validación independiente del conjunto de entrenamiento (que utilizamos como sustituto del error de generalización).

De hecho, ni siquiera utilizaremos directamente el error sobre el conjunto de entrenamiento como función objetivo de nuestro problema de optimización. Ese error, también conocido como pérdida 0-1 [*0-1 loss*], haría intratable nuestro problema de optimización desde el punto de vista computacional. La función de pérdida 0-1 es una función discreta: 0 para un ejemplo si éste se clasifica correctamente, 1 cuando se comete un error. En sustitución de la pérdida 0-1, se suelen emplear otras funciones cuya optimización resulte más sencilla, como el error cuadrático medio o la verosimilitud [*likelihood*]. En el caso de la verosimilitud, se estima la probabilidad condicional de una clase dada una entrada x : $p(y|x)$. El negativo del logaritmo de la verosimilitud, $-\log p(y|x)$, se puede utilizar entonces en sustitución de la función de pérdida 0-1.

Aunque pueda parecer poco intuitivo a primera vista, el uso de funciones de error sustitutas [*surrogate loss function*], además de resolver los problemas computacionales asociados al error 0-1, puede contribuir a que aprendamos mejor. Por ejemplo, después de conseguir reducir a cero la función de pérdida 0-1 sobre el conjunto de entrenamiento (100 % de acierto o, si lo prefiere, 0 % de tasa de error), el entrenamiento de una red puede continuar si utilizamos la verosimilitud, que se puede seguir reduciendo iterativamente. De esta forma, se consiguen modelos más robustos, separando mejor unas clases de otras y extrayendo más

información del conjunto de entrenamiento de la que podríamos haber obtenido si estuviésemos minimizando directamente la función de pérdida 0-1.

En definitiva, al plantear un problema de aprendizaje como un problema de optimización, convertimos el proceso de entrenamiento de un modelo en un proceso iterativo de optimización. En vez de detener este proceso iterativo en el momento en el que llegamos a un óptimo local de la función de error utilizada (p.ej. verosimilitud), como haríamos para resolver problemas matemáticos de optimización, lo detenemos en el momento en el que así lo indique un criterio de parada como *early stopping*. Este criterio de parada sí que lo podemos definir sobre la función de pérdida que realmente nos interesa optimizar (el error 0-1 sobre el conjunto de validación cuando usamos *early stopping*), de forma que seamos capaces de prevenir el sobreaprendizaje.

El caso unidimensional

En el entrenamiento de una red neuronal, pretendemos minimizar una función de error que depende del conjunto de parámetros de la red. Dado que una red incluye muchos parámetros, por lo que hemos de resolver un problema de optimización multivariable, tal vez le sorprenda que empecemos nuestro recorrido por las técnicas numéricas de optimización para funciones de una sola variable.

En realidad, tal como mencionamos en la introducción, muchas técnicas de optimización se basan en seleccionar primero una dirección en la que optimizar una función y, a continuación, determinar cuál es el cambio idóneo en esa dirección que nos permite minimizar la función de error. Se trata de las técnicas de búsqueda lineal. Si tenemos una función de error, coste o pérdida $f(x)$ definida sobre un espacio multidimensional de n dimensiones, podemos partir de un punto p en ese espacio multidimensional (los valores actuales de los parámetros de nuestro modelo) y movernos en una dirección d . Podemos minimizar nuestra función de n variables a lo largo de la línea d resolviendo un problema de optimización unidimensional:

$$\delta^* = \arg \min_{\delta} f(p + \delta d)$$

Dados los vectores que determinan el punto de partida p y la dirección d , nuestro problema consiste en encontrar el escalar δ que minimiza la función $f(p + \delta d)$. Cualquier método de optimización unidimensional nos puede ayudar a resolver un problema de este tipo.

Para resolver un problema de minimización de una función definida sobre una variable escalar, que asumiremos continua, lo primero que tenemos que hacer es determinar un intervalo en el que tengamos garantías de que se encuentre un mínimo de la función. Si estuviésemos buscando

una raíz de la función (un cero), nos bastaría con encontrar dos puntos a y b en los que la función tuviese distinto signo para saber que la raíz estará en el intervalo $[a, b]$. Para determinar la existencia de un mínimos, tendremos que encontrar una tripleta de puntos, $a < b < c$, tal que el valor de la función en el punto intermedio, $f(b)$, sea menor que en los extremos, $f(a)$ y $f(c)$. Entonces sabremos que la función incluye un mínimo en el intervalo $[a, c]$.

¿Cómo acotamos el intervalo en el que se encuentra un mínimo? Tenemos que encontrar un intervalo que incluya algún punto b tal que $f(b)$ sea menor que los extremos $f(a)$ y $f(c)$. Partimos de dos puntos a y b , con $f(a) > f(b)$. A continuación, utilizamos la razón áurea $\varphi = (1 + \sqrt{5})/2 \approx 1.618034$ para ampliar el tamaño de nuestro intervalo $[a, b]$ seleccionando un tercer punto: $c = b + \varphi(b - a)$. Si $f(b) < f(c)$, sabemos que habrá un mínimo en el intervalo $[a, c]$. Si no, sabemos que $f(a) > f(b) > f(c)$. Realizamos una extrapolación parabólica a partir de a , b y c . Esto es, determinamos el punto u en el que se halla el mínimo de la parábola que pasa por los tres puntos a , b y c . Cuando la parábola tiene su mínimo en un punto u entre b y c , comprobamos si $f(u) < f(c)$. Si es así, habrá un mínimo en el intervalo $[b, c]$. Si no, comprobamos también si $f(b) < f(u)$, situación en la que el mínimo estaría en el intervalo $[a, u]$. Si esto no nos ayuda a acotar el intervalo en el que se halla un mínimo de la función, repetimos el proceso descartando el punto más antiguo (a) y ampliando el tamaño del intervalo. Partiendo del intervalo $[b, c]$, en el que $f(b) > f(c)$, lo ampliamos con $u = c + \varphi(c - b)$... y así sucesivamente hasta ser capaces de determinar un intervalo en el que la función $f(x)$ incluya algún mínimo.

Una vez que sabemos que la función $f(x)$ tiene un mínimo dentro de un intervalo $[a, c]$, procedemos a localizar el punto en el que se encuentra ese mínimo utilizando un algoritmo divide y vencerás:

Búsqueda ternaria

A diferencia de la búsqueda de raíces, la búsqueda de mínimos involucra siempre el uso de tripletas de puntos, por lo que podemos utilizar un algoritmo análogo a la bisección o a la búsqueda binaria. En problemas de optimización, la búsqueda ternaria [*ternary search*] elige dos puntos intermedios m_1 y m_2 que dividen el intervalo $[a, c]$ en tres intervalos del mismo tamaño: $m_1 = a + (c - a)/3$ y $m_2 = c - (c - a)/3$. En cada iteración del algoritmo, se comparan los valores de la función en los puntos intermedios. Si $f(m_1) < f(m_2)$, el mínimo estará en $[a, m_2]$. Si $f(m_1) > f(m_2)$, el mínimo estará en $[m_1, c]$. En cualquier caso, el tamaño del intervalo resultante es $2/3$ del tamaño del intervalo original. En cada iteración, vamos reduciendo del intervalo en el que se encuentra el mínimo hasta llegar a localizarlo con la precisión deseada.

Búsqueda de la sección áurea

Cada iteración de la búsqueda ternaria requiere evaluar la función f en dos puntos. Podemos reducir a la mitad el número de evaluaciones de la función f si utilizamos la siguiente estrategia. Para acotar el intervalo en el que se encuentra el mínimo, ya utilizamos tres puntos (a, b, c) tales que $f(a) > f(b) < f(c)$. Podemos seleccionar un punto x entre b y c para distinguir dos casos posibles. Si $f(b) < f(x)$, el mínimo estará en $[a, x]$ por lo que repetiremos el proceso utilizando la tripleta (a, b, x) . Si $f(b) > f(x)$, el mínimo estará en $[b, c]$, por lo que la tripleta pasará a ser (b, x, c) . De esta forma conseguimos reducir el tamaño del intervalo en el que se encuentra el mínimo empleando una única evaluación de la función, $f(x)$.

¿Cómo seleccionamos el valor x ? Supongamos que, en nuestra tripleta (a, b, c) , b está a una fracción w del camino entre a y c : $(b - a) = w(c - a)$. El siguiente punto en el que evaluaremos la función estará una fracción z más allá de b : $(x - b) = z(c - a)$. Una vez evaluada la función $f(x)$, el intervalo en el que tendremos que continuar la búsqueda será de tamaño $w + z$ (si el mínimo queda en $[a, x]$) o de tamaño $1 - w$ (si el mínimo queda en $[b, c]$). Para minimizar el tamaño del intervalo en el peor caso, hacemos que ambos intervalos sean del mismo tamaño: $w + z = 1 - w$. Es decir, $z = 1 - 2w$. El punto x debería estar en la misma posición en el intervalo $[b, c]$ que el punto b en el intervalo $[a, c]$, algo que podemos expresar como $z/(1 - w) = w$. Combinando las dos expresiones anteriores, obtenemos una ecuación cuadrática $w^2 - 3w + 1 = 0$ cuya solución es $w = (3 - \sqrt{5})/2 \approx 0.38197 = 2 - \varphi$. El punto x está a una distancia $1 - (\varphi - 1)$ de un extremo del intervalo o, lo que es lo mismo, a una distancia $\varphi - 1$ del otro extremo del intervalo. Este resultado explica el nombre del método [*golden-section search*] y la aparentemente extraña elección que hicimos del factor con el que íbamos ampliando el tamaño del intervalo mediante el cual acotamos, anteriormente, la posición del mínimo de la función $f(x)$.

En cada iteración del algoritmo, dada una tripleta (a, b, c) , se elige un punto intermedio en el intervalo más grande (medido desde el punto central b). De esa forma, el tamaño del intervalo se reduce en proporción a $\varphi - 1 \approx 0.61803$, más que con la búsqueda ternaria y realizando una única evaluación de la función objetivo. La convergencia del algoritmo es lineal, en el sentido de que se descubren dígitos adicionales de la posición donde se encuentra el mínimo.

El algoritmo iterativo termina cuando $|c - a| < \tau(|b| + |x|)$, donde τ indica la tolerancia del algoritmo numérico (p.ej. $2.0\text{e-}4$ para precisión simple, con números en coma flotante de 32 bits, o $3.0\text{e-}8$ para precisión doble, usando números de 64 bits). La tolerancia del algoritmo no se corresponde con la precisión de los números en coma flotante en el

La razón áurea, φ en honor al escultor griego Fidias, surge de la división en dos de un segmento en el que se conservan las proporciones de la longitud total con respecto al segmento más largo y del segmento más largo con respecto al segmento más corto: $(a + b)/a = a/b$. El astrónomo alemán Johannes Kepler descubrió que el límite de la razón entre dos números consecutivos de la sucesión de Fibonacci es, precisamente, la razón áurea.

ordenador porque la función $f(x)$, cerca del mínimo b , se puede aproximar, utilizando una serie de Taylor, por $f(x) \approx f(b) + \frac{1}{2}f''(b)(x - b)^2$, al ser $f'(b) = 0$. El segundo término será despreciable siempre que sea un factor ϵ más pequeño en comparación con el primero, de donde se obtiene

$$|x - b| < \sqrt{\epsilon} |b| \sqrt{\frac{2|f(b)|}{b^2 f''(b)}}$$

donde ϵ corresponde a la precisión en coma flotante del ordenador. La última raíz cuadrada será de orden unitario habitualmente, por lo que no tiene demasiado sentido intentar imponer una tolerancia menor que $\sqrt{\epsilon}$ en la práctica.

Interpolación parabólica inversa

Una alternativa al método de la sección áurea es la interpolación parabólica inversa [*inverse parabolic interpolation*]: interpolación parabólica porque se calcula una parábola que pasa por tres puntos ($f(a)$, $f(b)$ y $f(c)$) e inversa porque no estamos interesados en calcular el valor de la función, sino en su inversa para determinar el punto x donde la parábola con la que aproximamos $f(x)$ alcanza su mínimo.

Dada una tripleta de puntos (a, b, c) y los valores de la función f para esos puntos $(f(a), f(b), f(c))$, la fórmula que nos da el valor x de la abscisa correspondiente al mínimo de la parábola que pasa por $f(a)$, $f(b)$ y $f(c)$ es la siguiente:

$$x = b - \frac{1}{2} \frac{(b-a)^2[f(b) - f(c)] - (b-c)^2[f(b) - f(a)]}{(b-a)[f(b) - f(c)] - (b-c)[f(b) - f(a)]}$$

La fórmula anterior funciona salvo que los tres puntos sean colineales, caso en el que el denominador será cero y no podremos utilizarla. A diferencia del método de la sección áurea, que nos permitía ir acotando paulatinamente el intervalo en el que sabemos que se encontrará el mínimo, la interpolación parabólica inversa nos permitiría llegar al mínimo de la función en un solo paso siempre que la función sea cuadrática en el entorno del mínimo, una aproximación que puede ser razonable una vez acotado correctamente el intervalo $[a, c]$.

Obviamente, la función real que queremos optimizar no será una parábola perfecta, por lo que podríamos intentar diseñar un algoritmo iterativo basado en la interpolación parabólica inversa para localizar su mínimo. Sin embargo, aplicar directamente la fórmula anterior no funcionará bien, ya que fallará cuando los puntos sean colineales y también cuando el mínimo x coincida exactamente con uno de los tres puntos de la tripleta (a, b, c) , lo que haría cero el denominador de la fórmula en la siguiente iteración del algoritmo.

En la práctica, recurriremos al método de Brent, que aprovecha la interpolación parabólica inversa cuando es posible y recurre a una estra-

De acuerdo con el estándar IEEE 754, la precisión de un número en coma flotante de precisión simple, de 32 bits, es del orden de 2^{-24} ($\approx 6 \cdot 10^{-8}$), mientras que es de 2^{-53} ($\approx 10^{-16}$) para los números de precisión doble, usando 64 bits.

tegía “divide y vencerás”, como el método de la sección áurea, más lenta pero segura, cuando no lo es.

El método de Brent

El matemático australiano Richard Brent diseñó un método de optimización⁴⁴² basándose en el trabajo previo del matemático holandés Theodorus Dekker, por lo que el método de Brent también se conoce como método de Brent-Dekker.

El algoritmo de Brent es un algoritmo iterativo de minimización que combina el método de la bisección (usando secciones áureas) con el método de la secante (usando una interpolación cuadrática en vez de la interpolación lineal típica del método de la secante). Esto es, combina el método de la sección áurea con la interpolación parabólica inversa. De esta forma, resulta tan fiable como el método de la bisección y tan rápido como sus alternativas más eficientes pero menos fiables.

En cada etapa, el algoritmo de Brent mantiene un conjunto de seis puntos, no necesariamente distintos: a , b , u , v , w y x . El significado de esos puntos es el siguiente: el intervalo $[a, b]$ contiene al mínimo de la función, x es el punto con el menor valor de la función conocido hasta el momento (o el más reciente en caso de empate), w corresponde al segundo menor valor, v es el valor previo de w y u es el punto en el que la función se evaluó más recientemente. En el algoritmo también se utiliza el punto intermedio entre a y b , x_m , aunque la función no se evalúa en ese punto.

En primer lugar, se intenta realizar una interpolación parabólica inversa a través de los puntos $f(x)$, $f(v)$ y $f(w)$. Esa interpolación sólo se acepta si el mínimo de la parábola cae dentro del intervalo $[a, b]$ e implica una mejora del mejor valor actual x que sea al menos la mitad que la mejora obtenida en la iteración anterior. Esta segunda condición nos permite asegurar que el método está convergiendo en dirección al mínimo en vez de oscilar. En el peor caso posible, cuando las interpolaciones parabólicas no se consideren aceptables, el método combinará pasos de la interpolación parabólica con iteraciones del método de la sección áurea, que es el que garantiza la convergencia del algoritmo. Además, la convergencia del algoritmo será superlineal, gracias al uso de la interpolación parabólica inversa.

Así pues, cuando desee minimizar una función univariable sin necesidad de calcular derivadas, sólo tiene que acotar la posición del mínimo (usando un algoritmo iterativo basado en la razón áurea) y aplicar el método de Brent a continuación. Si la función que desea minimizar tiene una segunda derivada discontinua, la interpolación parabólica del método de Brent no le ofrece ventaja alguna y puede usar el método más simple de la sección áurea.

⁴⁴²Richard P. Brent. *Algorithms for Minimization without Derivatives*. Prentice-Hall, 1973. ISBN 0130223352
Dekker es más conocido en Informática por inventar la primera solución conocida correcta al problema de la exclusión mutua en programación concurrente, lo que permite a dos hebras compartir un recurso de uso exclusivo utilizando memoria compartida.

¿Podemos usar el valor de la derivada de la función?

Dada una tripleta de puntos (a, b, c) , si disponemos de la capacidad de calcular la derivada de la función además del valor de la función, podemos intentar aprovechar esa capacidad para acotar mejor la posición del mínimo de la función.

Una posibilidad sería utilizar los valores de la función y su derivada para interpolar un polinomio cúbico o de orden superior, aunque una vez que tenemos una convergencia superlineal con el método de Brent probablemente no apreciemos la diferencia. Además, la convergencia superlineal se consigue sólo cuando conseguimos acercarnos al mínimo. Sin embargo, el uso de polinomios de orden superior puede complicarnos la vida innecesariamente, ya que sus interpolaciones pueden oscilar bruscamente y resultar completamente erróneas.

Sin embargo, sí que podemos hacer un uso (limitado) de la derivada de la función para diseñar una variante del método de Brent. El signo de la derivada en el punto central de la tripleta (a, b, c) nos indica de forma unívoca si el siguiente punto que debemos considerar está en el intervalo $[a, b]$ o en el intervalo $[b, c]$. El valor de esa derivada y el de la derivada en el segundo mejor punto conocido se puede extrapolar a cero utilizando el método de la secante (una interpolación lineal inversa), que es superlineal de orden φ (otra vez la razón áurea). Es decir, podemos sustituir la interpolación cuadrática inversa a partir de tres puntos de la función f por una interpolación lineal inversa realizada a partir de dos evaluaciones de la derivada f' . El resto del algoritmo funciona de la misma forma: si el punto considerado no satisface las restricciones impuestas por el método de Brent, se descarta la interpolación y se bisecciona el intervalo.

En cualquier caso, el cálculo de derivadas puede resultar problemático desde el punto de vista numérico. Integrar las derivadas calculadas numéricamente puede que no siempre dé como resultado el valor de la función. Del mismo modo, una derivada calculada numéricamente puede que no apunte adecuadamente en dirección hacia el mínimo, por lo que hemos de ser extremadamente cautelosos si decidimos utilizar un método numérico de optimización que utilice el valor de las derivadas de la función que pretendemos minimizar.

El caso multidimensional

Tras haber repasado las técnicas que nos permiten encontrar el mínimo de funciones de una sola variable, $f(x) : \mathbb{R} \rightarrow \mathbb{R}$, analicemos con mayor detenimiento las técnicas de optimización que nos permiten localizar mínimos en funciones multivariable, $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$.

En el caso multivariable, una función recibe como parámetro un vector $x = (x_1, x_2, \dots, x_n)$. Otra cantidad vectorial, el gradiente, desempeña el

papel de la derivada en el caso unidimensional. El gradiente se suele representar por el símbolo “nabla”, también conocido como “del”: ∇ . Este símbolo no es más que la letra griega delta invertida, $\Delta \rightarrow \nabla$, y su nombre proviene de la palabra griega $\nu\alpha\beta\lambda\alpha$, que significa “arpa”, un instrumento musical con una forma similar a la del símbolo. Matemáticamente, el operador vectorial ∇ se obtiene a partir de las derivadas parciales de una función con respecto a sus parámetros:

$$\nabla_x = \left[\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \dots, \frac{\partial}{\partial x_n} \right]$$

La notación vectorial nos ayuda a simplificar el aspecto de las expresiones con las que trabajaremos. El subíndice x lo suprimiremos cuando no exista ambigüedad con respecto a qué conjunto de variables aplicamos el gradiente.

Si aplicamos el operador ∇ a una función $f(x)$, el gradiente $\nabla f(x)$ nos indica la dirección de máxima pendiente de la función en el punto x :

$$\nabla_x f(x) = \left[\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \dots, \frac{\partial f(x)}{\partial x_n} \right]$$

La dirección en la que la función $f(x)$ decrece más rápidamente es, precisamente, la opuesta al gradiente, de ahí que el método del gradiente descendente actualice los valores de los parámetros de una función de acuerdo a la siguiente expresión para tratar de acercarse a un mínimo de la función:

$$\Delta x = -\eta \nabla f(x)$$

El gradiente descendente es, en espacios continuos, el equivalente a un sencillo algoritmo de ascensión de colinas [*hill climbing*] en espacios de búsqueda discretos. El cálculo del gradiente suele facilitar la resolución de problemas de optimización en espacios multidimensionales y, por este motivo, recurren a él casi todos los métodos que se utilizan en la práctica para entrenar redes neuronales (i.e., ajustar sus parámetros con el objetivo de minimizar una función de coste o pérdida predefinida).

Sin embargo, no todos los métodos de optimización que se han propuesto utilizan el gradiente:

- *El método de Nelder-Mead*

El algoritmo de optimización propuesto por los británicos John Ashworth Nelder y Roger Mead⁴⁴³ sólo necesita evaluar la función, sin necesidad de derivadas. Se basa en el concepto de *simplex*, un politopo de $n + 1$ vértices en un espacio de n dimensiones. Un politopo no es más que la generalización de un polígono bidimensional o un poliedro tridimensional a un espacio multidimensional. Por tanto, un segmento de línea en un espacio unidimensional, un triángulo en el plano, un tetraedro en el espacio y pentácoron en un espacio de 4 dimensiones son ejemplos de simplices.

⁴⁴³ John A. Nelder y Roger Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965. doi: 10.1093/comjnl/7.4.308

El método de Nelder-Mead busca un óptimo local de una función multidimensional actualizando progresivamente los vértices del *simplex*. En cada iteración, el algoritmo puede reflejar el punto superior del *simplex* (el vértice del simplex donde la función toma un valor mayor) a través de la cara opuesta del *simplex* hasta obtener un punto inferior, conservando el volumen del *simplex*. También puede expandir el simplex, aumentando su volumen, o contraerse alrededor de su punto más bajo (el mejor desde el punto de vista de la función de optimización).

El algoritmo funciona sin realizar suposiciones acerca de la función que se desea optimizar, por lo que es extremadamente robusto. Sin embargo, puede ser demasiado lento y requiere un espacio de memoria cuadrático con respecto al número de dimensiones del problema ($n + 1$ puntos de n dimensiones), por lo que no resulta adecuado para entrenar redes neuronales, donde podemos tener millones de parámetros que ajustar (millones de dimensiones).

■ *El método de Powell*

El algoritmo de optimización propuesto por el también británico Michael J.D. Powell⁴⁴⁴ es mucho más rápido que el método de Nelder-Mead. En este caso, el algoritmo se basa en seleccionar un conjunto de direcciones mutuamente conjugadas y aplicar un algoritmo de optimización unidimensional, como el método de Brent, en cada una de las direcciones conjugadas.

Dos direcciones son conjugadas si la optimización realizada en una de ellas no interfiere en la optimización realizada en la otra: la minimización de la función a lo largo de una dirección no se estropea al minimizar la función a lo largo de una dirección conjugada. En otras palabras, lo que hacemos en una dirección no deshace lo que ya hayamos conseguido en otra dirección, de forma que se evitan ciclos interminables y el algoritmo resultante converge.

En el caso del método de Powell, se inicializa el conjunto de las direcciones u_i con los n vectores base del espacio multidimensional. Comenzando desde el punto inicial p_0 , se minimiza la función a lo largo de la dirección u_i para obtener el punto p_i . Tras hacer esto con los n vectores u_i , se descarta el primero, desplazando los demás una posición a la izquierda, y se añade un vector $u_n = p_n - p_0$, tras lo que se mueve el punto p_n hasta el mínimo a lo largo de la dirección u_n para obtener un nuevo punto p_0 . Tras n iteraciones de este algoritmo, que involucran $n(n + 1)$ búsquedas lineales, se consigue minimizar de forma exacta cualquier función cuadrática.

Obviamente, el método sigue siendo demasiado ineficiente cuando tenemos millones de variables, tanto en espacio, $O(n^2)$ para almacenar n vectores de n dimensiones, como en tiempo, $O(n^2)$ búsquedas lineales. Por este motivo, para entrenar redes neuronales artificiales siempre

⁴⁴⁴ M. J. D. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal*, 7(2):155–162, 1964. DOI: 10.1093/comjnl/7.2.155

recurriremos al uso del gradiente y dejaremos este tipo de métodos de optimización para situaciones en las que no podamos calcular derivadas.

Así pues, a diferencia de lo que sucedía en el caso unidimensional, en el caso multidimensional hemos llegado a la conclusión de que necesitamos utilizar las derivadas de la función para resolver problemas de optimización como los que nos encontramos al entrenar redes neuronales artificiales.

Para obtener el gradiente de la función que deseamos minimizar podemos, o bien calcularlo de forma analítica utilizando su expresión matemática (de ahí la importancia que tenía para nosotros seleccionar funciones de activación no lineales que fuesen diferenciables) o bien estimarlo tomando diferencias finitas entre valores de la función (un método mucho más inestable y propenso a errores de tipo numérico). Usualmente, optaremos siempre por la primera opción.

Volvamos una vez más al gradiente descendente propuesto por Cauchy en 1847, que nos servirá de base para obtener diferentes algoritmos de optimización:⁴⁴⁵

$$\Delta x = -\eta \nabla_x f(x)$$

En la expresión anterior, el parámetro η es la tasa de aprendizaje que determina el tamaño de los saltos que vamos dando en cada iteración del algoritmo. Esta tasa de aprendizaje se puede fijar de distintas formas:

- *Tasa de aprendizaje constante*: Seleccionamos un valor pequeño, que no ocasione problemas de convergencia, para ir acercándonos al óptimo de la función pero sin pasarnos. Es la estrategia más simple pero también la más ineficiente.
- *Tasas de aprendizaje adaptativas*: Seleccionamos una tasa de aprendizaje inicial, que vamos ajustando paulatinamente conforme progresa la ejecución del algoritmo iterativo de optimización. Al estudiar el entrenamiento de redes neuronales multicapa ya vimos algunas formas de establecer tasas locales de aprendizaje, como Quickprop o el uso factores de ganancia.
- *Solución analítica*: En situaciones muy puntuales, tal vez podamos resolver de forma analítica cuál es el valor x que hace que el gradiente $\nabla_x f(x)$ desaparezca, lo que nos permitirá saltar directamente al punto crítico de la función $f(x)$.
- *Búsqueda lineal [line search]*: Una vez que el gradiente nos indica la dirección en la que movernos, podemos recurrir a técnicas de optimización unidimensional, como el método de Brent, para encontrar, a lo largo de la dirección dada por $-\nabla_x f(x)$, cuál es el valor escalar δ que minimiza la función $f(x - \delta \nabla_x f(x))$. Esta búsqueda lineal,

⁴⁴⁵ Augustin-Louis Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Compte Rendu des Séances de L'Académie des Sciences XXV*, Série A(25):536–538, 1847

obviamente, involucra la evaluación de la función $f(x - \delta \nabla_x f(x))$ para distintos valores de δ .

Además del gradiente descendente y todas sus variantes, existen otros métodos de optimización multivariable que también podemos utilizar para entrenar redes neuronales artificiales:

- *Gradientes conjugados:* Como en el método de Powell, podemos ir seleccionando direcciones conjugadas a la hora de resolver un problema de optimización usando gradientes. Los métodos que utilizan gradientes conjugados, como el algoritmo de Fletcher-Reeves o el algoritmo de Polak-Ribière, se utilizan bastante gracias a que requieren un espacio de almacenamiento lineal, $O(n)$, aprovechan la información que nos ofrece el gradiente y se basan en la resolución de problemas de minimización unidimensional.
- *Técnicas de optimización de segundo orden:* El método de Newton utiliza la segunda derivada de la función para resolver en un solo paso problemas de optimización cuadrática. En el caso multidimensional, la segunda derivada viene dada por la matriz hessiana, una matriz cuadrada, de tamaño $n \times n$, que contiene las segundas derivadas parciales de la función $f(x)$. Cuando resulta demasiado costoso calcular la matriz hessiana en cada iteración del algoritmo, se utilizan métodos quasi-Newton como DFP [Davidon-Fletcher-Powell] o BFGS [Broyden-Fletcher-Goldfarb-Shanno]. Estos métodos utilizan las derivadas parciales de la función y métodos unidimensionales de minimización, si bien requieren un espacio de almacenamiento cuadrático, $O(n^2)$, lo que los hace inviables cuando el número de parámetros de la red es elevado. Por este motivo, se han diseñado variantes de estos algoritmos que hacen un uso limitado de memoria, como L-BFGS [*Limited-memory BFGS*].

Antes de estudiar con mayor detalle todos estos algoritmos de optimización, demos un paso atrás y analicemos el tipo de problema de optimización al que nos enfrentamos cuando deseamos ajustar los parámetros de una red neuronal artificial.

Mínimos locales y otros puntos críticos

El entrenamiento de una red neuronal lo hemos convertido en la resolución un problema matemático de optimización en un espacio multidimensional. Este espacio, que puede contener millones de dimensiones, requiere disponer de técnicas de optimización que, además de ser escalables, sean capaces de trabajar con los múltiples óptimos locales asociados al entrenamiento de una red.

La peculiar arquitectura de una red neuronal hace que necesariamente existan múltiples óptimos locales en la función que pretendemos optimizar:

- En una red neuronal multicapa, podemos intercambiar las posiciones relativas de las neuronas de una capa oculta (y los pesos asociados a las sinapsis que las conectan con el resto de la red) y obtener redes completamente equivalentes. En una capa completamente conectada de k neuronas ocultas, existen $k!$ permutaciones que nos darán como resultado modelos idénticos. Esta característica de las redes neuronales se conoce con el nombre de simetría en el espacio de pesos [*weight space symmetry*].
- También podemos escalar los pesos de entrada de una neurona por un factor c y los de salida por un factor $1/c$ para obtener redes equivalentes si en nuestra función de coste no se incluyen términos de regularización en los que intervengan los valores de los pesos de la red, como sucede al usar *weight decay*.

Si estuviésemos ante un problema de optimización convexa, no tendríamos problema alguno al utilizar el gradiente descendente. Tendríamos garantías de, eventualmente, llegar al mínimo global de nuestra función de coste.

Sin embargo, al entrenar redes neuronales artificiales, nos enfrentamos a problemas de optimización no convexa, en los que existen múltiples mínimos locales. El mínimo al que lleguemos al utilizar métodos basados en el gradiente descendente dependerá de nuestra posición inicial sobre la superficie de error y de los hiperparámetros que utilicemos para el algoritmo concreto de optimización que estemos utilizando.

La existencia de múltiples óptimos locales está relacionada con el problema de la identificabilidad de un modelo. Un modelo se dice identificable cuando un conjunto de entrenamiento lo suficientemente grande es capaz de descartar todas las posibles configuraciones de sus parámetros menos una. En el caso de las redes neuronales, su simetría en el espacio de pesos hace que sean modelos intrínsecamente no identificables. Esto nos obliga a trabajar con funciones que tienen múltiples óptimos locales.

No obstante, muchos de los mínimos locales de la función de coste con la que ajustamos los parámetros de una red neuronal son, realmente, equivalentes unos a otros, por lo que no resultan problemáticos desde el punto de vista del problema de aprendizaje. El hecho de que la optimización no sea convexa no supone ningún impedimento para entrenar correctamente una red neuronal.

Los mínimos locales sí pueden ser problemáticos cuando su coste difiere notablemente del óptimo global de la función de coste. En ese caso, puede que una técnica de optimización basada en el gradiente descendente se

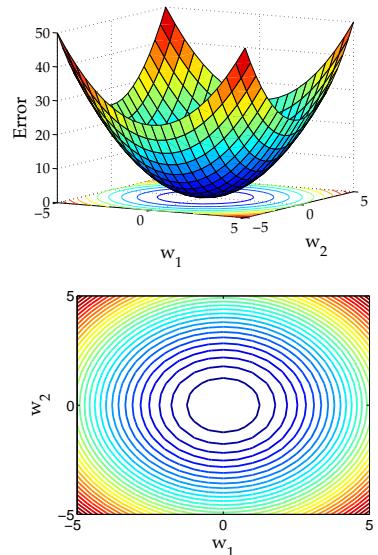


Figura 143: Problema de optimización convexa: La función $f(w) = w_1^2 + w_2^2$ tiene un único mínimo global.

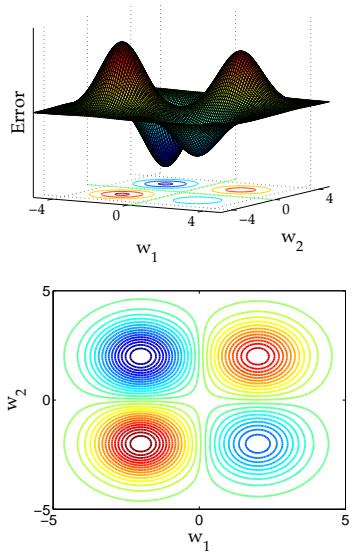


Figura 144: Múltiples óptimos locales: En función de nuestro punto de partida, llegaremos a uno de los dos mínimos locales de la función de error, sin garantías de encontrar el mínimo global.

quede atascada en un óptimo local muy alejado del óptimo global y los resultados obtenidos con el entrenamiento de la red neuronal sean, digamos, subóptimos. Ésta es una de las causas por las que las redes neuronales pasaron a segundo plano a finales de los años 90: en un problema de optimización no convexa con un número exponencial de mínimos locales, no existen garantías reales de obtener una solución óptima.

Sin embargo, en los últimos años, esa percepción negativa ha cambiado. Desde el punto de vista del gradiente descendente, lo importante son los puntos críticos de la función $f(x)$, aquéllos para los que su gradiente $\nabla f(x)$ se anula. Los puntos críticos corresponden a zonas planas de la superficie de error, sin apenas gradiente, por lo que pueden resultar problemáticas en problemas iterativos de optimización que usan el gradiente para modificar los valores de los parámetros de la red.

En pocas dimensiones, pueden existir múltiples mínimos locales de los que es difícil escapar usando el gradiente descendente. Sin embargo, en espacios de muchas dimensiones, los mínimos locales no son los puntos críticos más comunes, sino los puntos de silla [*saddle points*]. En los puntos de silla, la curvatura de la superficie de error es positiva en algunas direcciones y negativa en otras. A diferencia de lo que sucede con los mínimos locales, las técnicas de optimización evitan fácilmente quedarse atascadas en los puntos de silla.⁴⁴⁶

En un mínimo de la función $f(x)$, la curvatura de su superficie es positiva en todas direcciones. En un máximo, es negativa en todas direcciones. En un punto de silla, sin embargo, en algunas direcciones la curvatura es positiva y en otras es negativa, lo que nos permitirá escapar del punto de silla en caso de llegar a él.

El índice de un punto crítico es el número o fracción de autovalores [*eigenvalues*] negativos de su matriz hessiana, la forma matemática de describir el número de direcciones en los que la curvatura de la superficie de la función es negativa: 0 para un mínimo, n para un máximo (1 si hablamos de fracciones) y un valor intermedio para los puntos de silla (que son máximos de la función en algunas direcciones y mínimos en otras). Formalmente, estamos ante un mínimo si la matriz hessiana es definida positiva (todos sus autovalores positivos), ante un máximo si la matriz hessiana es definida negativa (todos sus autovalores negativos) y ante un punto de silla si la matriz hessiana es indefinida (tanto con autovalores positivos como con autovalores negativos). Para familias de funciones bastante amplias, existe una distribución de probabilidad no uniforme para el índice de los puntos críticos.

Dada una función multivariante $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$, la proporción esperada de puntos de silla con respecto al número de mínimos locales crece exponencialmente con la dimensión n . Esto quiere decir que, cuando estamos intentando optimizar una función de dimensionalidad elevada,

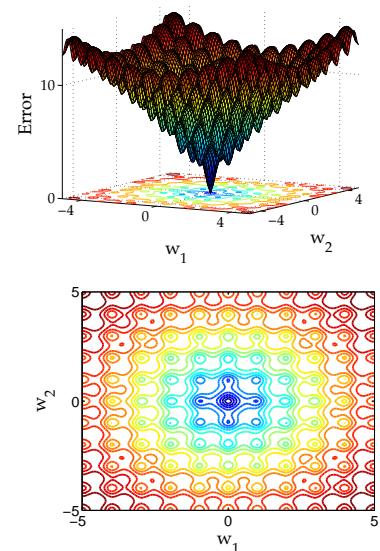


Figura 145: La función de Ackley: Un problema de optimización muy difícil, con múltiples óptimos locales pero un único óptimo global.

⁴⁴⁶ Ian J. Goodfellow, Oriol Vinyals, y Andrew M. Saxe. Qualitatively characterizing neural network optimization problems. *ICLR'2015*, abs/1412.6544, 2015b. URL <http://arxiv.org/abs/1412.6544>

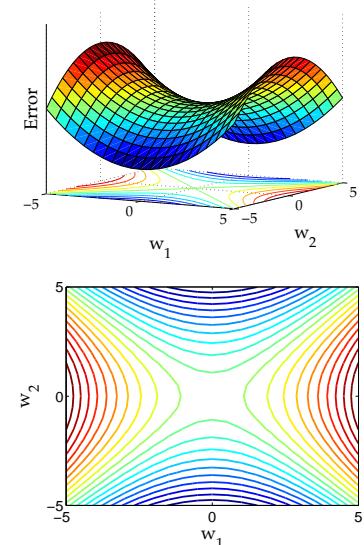


Figura 146: Punto de silla en $(0,0)$ de la función $f(w) = w_1^2 - w_2^2$. La curvatura de la superficie de error es positiva a lo largo de la silla (dimensión w_1) y negativa de un lado a otro (dimensión w_2).

como en el entrenamiento de redes neuronales, los puntos críticos con los que nos encontraremos serán, muy probablemente, puntos de silla. Si nos detenemos un momento a pensar, resulta natural que el número de puntos de silla sea superior al de mínimos locales. Para que un punto crítico sea un mínimo hace falta que la curvatura de la función sea positiva en todas direcciones: en n dimensiones la curvatura ha de tener el mismo sentido. En cuanto haya algunas direcciones para las que el sentido de la curvatura no coincide, nos encontraremos ante un punto de silla. Por tanto, que el signo de la curvatura coincida en todas direcciones y nos encontremos ante un mínimo resultará exponencialmente menos probable cuanto mayor sea el número n de dimensiones.^{447,448,449}

Así pues, aunque durante mucho tiempo se pensó que los mínimos locales suponían un grave problema para el entrenamiento de redes neuronales, resulta que no es así. Desde los años 80 se sabe que las funciones de coste con las que se entrena las redes neuronales tienen un número exponencial de mínimos locales (recordemos, entre otras causas posibles, la simetría en el espacio de pesos). Esa aparente debilidad de las redes neuronales, junto con el éxito de las máquinas de vectores de soporte, SVM, que sí pueden ofrecer garantías formales, hizo que decayese el interés en redes neuronales artificiales a finales de los 90.

Sin embargo, no se pueden atribuir todos los problemas del entrenamiento de redes neuronales a la presencia de mínimos locales. Una sencilla prueba nos puede ayudar a descartar los puntos críticos (mínimos y puntos de silla) como causa del rendimiento inadecuado de una red neuronal artificial. Si representamos gráficamente la magnitud del gradiente a lo largo del entrenamiento de una red, es habitual observar que la magnitud del gradiente no desaparece hasta hacerse insignificante conforme avanzan las épocas de entrenamiento de la red. De hecho, puede hasta aumentar durante el entrenamiento de la red. Este fenómeno descarta los puntos críticos como causa de los problemas en el entrenamiento de la red. A menudo, cuando se entrena una red, ni siquiera se llega a visitar mínimos locales, puntos de silla o mesetas en las que la superficie de error sea prácticamente plana. Aun cuando el gradiente descendente no llegue a punto crítico alguno e incluso aumente la magnitud del gradiente, si que se suele observar habitualmente cómo disminuye el error sobre el conjunto de validación, señal inequívoca de que el proceso de entrenamiento de la red funciona razonablemente bien.

La dimensionalidad elevada del problema de optimización asociado al entrenamiento de una red neuronal es, pues, un arma de doble filo. Por un lado, cuanto mayor sea la dimensionalidad del problema, más espacio hay para la presencia de óptimos locales y otros puntos críticos. Por otro lado, para que nos quedemos atrapados en un óptimo local, tenemos que quedarnos atrapados en todas las direcciones posibles, lo que hace realmente improbable que algo así llegue a suceder. Es decir, aunque el

⁴⁴⁷ Razvan Pascanu, Yann N. Dauphin, Surya Ganguli, y Yoshua Bengio. On the saddle point problem for non-convex optimization. *arXiv e-prints*, arXiv:1405.4604, 2014. URL <http://arxiv.org/abs/1405.4604>

⁴⁴⁸ Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, y Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger, editores, *NIPS'2014 Advances in Neural Information Processing Systems 27*, pages 2933–2941. Curran Associates, Inc., 2014b. URL <https://goo.gl/dE72X5>

⁴⁴⁹ Anna Choromanska, Mikael Henaff, Michaël Mathieu, Gérard Ben Arous, y Yann LeCun. The loss surface of multilayer networks. *AISTATS'2015*, arXiv:1412.0233, 2015. URL <http://arxiv.org/abs/1412.0233>

gradiente descendente sea un simple proceso de ascensión de colinas (o, más bien, de descenso al fondo del valle), en un espacio multidimensional nos encontramos muchos más puertos de montaña (puntos de silla) que cimas (óptimos locales). Esto hace que una estrategia aparentemente simple, como el gradiente descendente combinado con *backpropagation*, se haya utilizado con éxito para resolver problemas que, hasta hace sólo unos años, requerían intervención humana.

Gradiente descendente estocástico

Desde el punto de vista formal, los algoritmos de optimización se diseñan bajo la suposición de que disponemos del gradiente exacto de la función que deseamos optimizar (o de su matriz hessiana, en el caso de los métodos de optimización de segundo orden). Sin embargo, esto no es así cuando se utiliza para entrenar un modelo en aprendizaje automático. Sólo disponemos de una muestra de datos, el conjunto de entrenamiento, a partir de la cual estimamos el gradiente. Si el conjunto de entrenamiento no es realmente representativo de la distribución real de los datos del problema que estamos modelando, esa estimación estará sesgada.

El gradiente utilizado para actualizar los valores de los parámetros de una red neuronal se puede calcular a partir del conjunto de entrenamiento completo (aprendizaje por lotes), a partir de una muestra del conjunto de entrenamiento (aprendizaje por minilotes) o, incluso, a partir de un único ejemplo de entrenamiento (aprendizaje *online*). Cuanto menor sea la muestra utilizada para el cálculo del gradiente, mayor será el error cometido en su estimación.

Dicho de una forma más eufemística, siempre trabajaremos con gradientes con ruido, por lo que las actualizaciones que realizamos de los parámetros de la red, en realidad, son de la forma

$$\Delta x = -\eta (\nabla_x f(x) + \xi)$$

donde la letra griega xi (ξ) indica el error en nuestra estimación del gradiente.

El uso de gradientes con ruido hace que los algoritmos de optimización que utilicemos ni siquiera puedan garantizar que vayamos a alcanzar un mínimo local de la función $f(x)$ en un tiempo razonable. No obstante, normalmente funcionan lo suficientemente bien como para encontrar un valor para el vector de parámetros x con el que se obtienen resultados útiles en la práctica.

En realidad, el hecho de que las estimaciones del gradiente con las que trabajemos incluyan ruido juega a nuestro favor. Es la idea básica que justifica el uso del gradiente descendente estocástico [*SGD: Stochastic Gradient Descent*], término que engloba tanto al aprendizaje por minilotes como al aprendizaje *online*. Además de resultar mucho más económico

calcular una aproximación del gradiente que intentar realizar una estimación más exacta, el gradiente descendente estocástico no interfiere con la convergencia del algoritmo en problemas de optimización convexa y, en el caso de la optimización no convexa, con un número exponencial de mínimos locales y puntos de silla, la presencia de error en la estimación realizada del gradiente ayuda a escapar con mayor facilidad de los puntos de silla de la función de coste.⁴⁵⁰ Una vez más, como sucede a menudo en Informática, una limitación se acaba vendiendo como una característica: “*It’s not a bug, it’s a feature!*”.

Si nos ponemos algo más exquisitos desde el punto de vista formal, otro enfoque que podemos utilizar para justificar por qué la aproximación realizada con el gradiente descendente estocástico funciona se basa en interpretarlo como un proceso de inferencia bayesiana. La inferencia bayesiana es un método probabilístico en el que se aplica el teorema de Bayes para actualizar la probabilidad de una hipótesis conforme se recaba más información (más evidencias). Todo algoritmo numérico de optimización se puede ver como una forma de realizar inferencia bayesiana. Un truco mediante el cual un bayesiano acérrimo puede convertir cualquier algoritmo de aprendizaje automático que funcione bien en un caso particular de modelo bayesiano, con lo que puede atribuir el éxito del algoritmo a las bondades del aprendizaje bayesiano. En el caso del gradiente descendente estocástico, se puede interpretar éste como una cadena de Markov que, bajo determinadas suposiciones, tiene una distribución estacionaria que es una aproximación de la distribución de probabilidad de los parámetros de la red.⁴⁵¹ Dicho de forma más clara, cuando se detiene el proceso iterativo de optimización, en realidad estamos tomando una muestra de esta distribución aproximada.

La interpretación bayesiana le atribuye un rol principal al algoritmo de optimización utilizado para entrenar la red neuronal, tan importante como la propia arquitectura de la red, una peculiaridad que diferencia a las redes neuronales de otros modelos de aprendizaje automático como las SVM. A priori, podríamos pensar que una red neuronal no es más que un modelo sobreparametrizado, que puede llegar a tener más parámetros ajustables que datos disponibles para su entrenamiento. Esto nos podría llevar a pensar, erróneamente, que la red será incapaz de generalizar correctamente. Sin embargo, el gradiente descendente estocástico no sólo tiende a converger hacia un óptimo local, sino que esta sesgado a favor de óptimos locales que tengan propiedades deseables a la hora de generalizar.

Esta interpretación probabilística resulta interesante porque nos permite razonar acerca de los hiperparámetros del algoritmo de optimización. Por ejemplo, en el caso de la tasa de aprendizaje, cuando su valor es elevado, la cadena de Markov asociada al SGD pasa a ser inestable hasta encontrar una zona amplia alrededor de un mínimo local que cubre una mayor área (se aumenta la varianza del proceso de optimización). Cuando

⁴⁵⁰ Rong Ge, Furong Huang, Chi Jin, y Yang Yuan. Escaping From Saddle Points - Online Stochastic Gradient for Tensor Decomposition. *arXiv e-prints*, arxiv:1503.02101, 2015. URL <http://arxiv.org/abs/1503.02101>

⁴⁵¹ Stephan Mandt, Matthew D. Hoffman, y David M. Blei. Stochastic gradient descent as approximate bayesian inference. *arXiv e-prints*, arXiv:1704.04289, 2017. URL <http://arxiv.org/abs/1704.04289>

El entrenamiento de una SVM consiste en resolver un problema de optimización convexa, por lo que el algoritmo de optimización que utilicemos puede influir en la eficiencia del entrenamiento, pero no en las características del modelo que se obtiene como resultado. El papel del algoritmo de optimización es siempre secundario.

se disminuye el valor de la tasa de aprendizaje, la cadena de Markov aproxima mínimos más estrechos hasta converger a una región reducida del espacio de pesos de la red (se aumenta el sesgo a favor de la región del espacio en la que ya nos encontramos). Otro hiperparámetro del gradiente descendente estocástico, el tamaño del minilote, también controla el tipo de región hacia la que converge el algoritmo: más amplia para minilotes pequeños y más reducida para minilotes más grandes.

La discusión sobre “mínimos planos” [*flat minima*], en regiones amplias del espacio con una superficie de error relativamente plana, y “mínimos pronunciados” [*sharp minima*] no es nueva,⁴⁵² aunque se realice con un propósito totalmente distinto.⁴⁵³ El gradiente descendente estocástico tiende a preferir unos u otros en función de los valores que utilicemos para sus hiperparámetros (en particular, la tasa de aprendizaje y el tamaño del minilote). Así pues, el propio proceso estocástico de optimización es el que permite que las redes neuronales utilizadas en *deep learning* generalicen correctamente.

Se ha observado que, en ausencia de técnicas de regularización, las redes neuronales profundas generalizan mejor con minilotes más pequeños, lo que parece indicar que un tamaño reducido de minilote sesga el aprendizaje hacia “mínimos planos” en vez de hacia “mínimos pronunciados”.⁴⁵⁴ Tal vez eso explique por qué, en la práctica, funciona tan bien el aprendizaje *online*, como caso extremo del aprendizaje por minilotes. Aunque no todo el mundo le atribuye el éxito a los “mínimos planos”.⁴⁵⁵ Las ventajas que ofrece SGD con respecto a la capacidad de generalización del modelo obtenido se pierden, en parte, cuando se emplean otras técnicas de optimización populares como AdaGrad, RMSProp o Adam: aunque resultan más eficientes a la hora de entrenar la red, tienden a generalizar peor que el gradiente descendente estocástico.⁴⁵⁶

Los detalles del proceso de optimización, por tanto, son clave en el éxito de las técnicas de *deep learning*. Para intentar garantizar que el gradiente descendente estocástico funcione correctamente, hemos de tener en cuenta dos factores adicionales:

- *Un factor global:*

El proceso iterativo de optimización de los parámetros de la red describe una trayectoria a lo largo de la superficie de error asociada a la función de coste o pérdida. Muchos de los problemas que pueden surgir a la hora de resolver un problema de optimización complejo pasan a segundo plano cuando existe una región del espacio de búsqueda conectada a una solución mediante un camino que se pueda seguir utilizando el gradiente descendente estocástico y somos capaces de inicializar los parámetros de la red con valores que caigan dentro de esta región. Esto sugiere que establecer adecuadamente los valores iniciales de la red puede desempeñar un papel esencial en el entrenamiento de

⁴⁵² Sepp Hochreiter y Juergen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997. DOI: 10.1162/neco.1997.9.1.1

⁴⁵³ Geoffrey E. Hinton y Drew van Camp. Keeping the neural networks simple by minimizing the description length of the weights. En *Proceedings of the 6th Annual Conference on Computational Learning Theory*, COLT '93, pages 5–13, 1993. ISBN 0-89791-611-5. DOI: 10.1145/168304.168306

⁴⁵⁴ Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, y Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *ICLR'2017*, abs/1609.04836, 2017. URL <http://arxiv.org/abs/1609.04836>

⁴⁵⁵ Laurent Dinh, Razvan Pascanu, Samy Bengio, y Yoshua Bengio. Sharp minima can generalize for deep nets. En Doina Precup y Yee Whye Teh, editores, *ICML'2017 Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1019–1028, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL <http://proceedings.mlr.press/v70/dinh17b.html>

⁴⁵⁶ Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, y Benjamin Recht. The Marginal Value of Adaptive Gradient Methods in Machine Learning. *arXiv e-prints*, arXiv:1705.08292, 2017. URL <http://arxiv.org/abs/1705.08292>

redes neuronales. Así pues, algunas vías abiertas de investigación se encaminan a conseguir buenas condiciones iniciales para los parámetros de la red,⁴⁵⁷ en lugar de realizar movimientos no locales durante el proceso de optimización.

- *Un factor local:*

En la práctica, trabajamos con una estimación con ruido y sesgada del gradiente que se utiliza para guiar el proceso de optimización de los parámetros de una red. Además, las redes neuronales multicapa pueden incluir zonas en las que la función de coste tiene una pendiente muy elevada, regiones conocidas como acantilados o precipicios [*cliffs*]. Ante zonas de este tipo, el gradiente puede ser tan grande que la actualización nos aleje demasiado de nuestra zona de interés, propiciando un salto a una región del espacio de búsqueda que tal vez no sea tan interesante como el fondo del precipicio del que nos encontramos al borde, con lo que estaríamos perdiendo parte del trabajo de optimización que ya hayamos realizado.

Afortunadamente, prevenir estos saltos desproporcionados es sencillo si recortamos el gradiente [*gradient clipping*]. Se trata de una técnica heurística que se utiliza en la práctica desde hace años y que consiste, o bien en limitar el gradiente elemento a elemento, o bien en limitar la norma del gradiente antes de utilizarlo para actualizar los parámetros de la red. Esto es, si $\|\nabla f(x)\| > v$, usamos $v \nabla f(x) / \|\nabla f(x)\|$ en la fórmula de actualización de los pesos de la red, aunque no se observan demasiadas diferencias si, simplemente, nos limitamos a recortar los componentes individuales del vector del gradiente. El recorte del gradiente introduce un sesgo adicional en el cálculo del gradiente que, empíricamente resulta útil a la hora de entrenar redes neuronales.

Recordemos que el gradiente nos indica la dirección que corresponde a la disminución más pronunciada de nuestra función objetivo en un entorno infinitesimal alrededor del punto correspondiente a los valores actuales de los parámetros de la red. Fuera de esta región infinitesimal, la función de coste puede que cambie de aspecto, motivo por el que el recorte del gradiente nos ayuda a evitar que recorramos más camino del que deberíamos en una sola iteración del gradiente descendente. Al fin y al cabo, el gradiente descendente es efectivo a la hora de entrenar redes neuronales, precisamente, por estar basado en ir realizando pequeños saltos, movimientos locales en el paisaje endiabladamente montañoso de la función de coste que pretendemos minimizar.

Promediado de Polyak

Los orígenes del gradiente descendente estocástico se remontan a 1951, cuando Herbert Robbins y Sutton Monro presentaron su método de aproximación estocástica: un algoritmo iterativo de optimización que

⁴⁵⁷ Andrew M. Saxe, James L. McClelland, y Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *ICLR'2014*, arXiv:1312.6120, 2014. URL <http://arxiv.org/abs/1312.6120>

El recorte del gradiente elemento a elemento no preserva la dirección del gradiente real, aunque sigue apuntando en dirección descendente.

permite calcular los ceros o los extremos de una función a partir de observaciones con ruido.⁴⁵⁸

Cuando el mínimo x^* de una función $f(x)$ no se puede calcular directamente, se puede aproximar de forma eficiente utilizando muestras aleatorias de una función con ruido $F(x, \xi)$, cuyo valor esperado es igual al del gradiente de la función que nos interesa, $\nabla f(x) = E[F(x, \xi)]$. Nuestro objetivo será averiguar dónde es cero ese gradiente. Para ello, el método de Robbins-Monro realiza una serie de aproximaciones de la misma forma que el gradiente descendente:

$$\Delta x = x_{n+1} - x_n = -\eta_n \nabla f(x_n)$$

donde $\{\eta_n\}$ es una secuencia de tamaños de paso, equivalentes a una tasa de aprendizaje que se va reduciendo conforme avanzan las iteraciones del algoritmo. Robbins y Monro sugerían utilizar $\eta_n = \eta/n$, donde n es la iteración del algoritmo y $\eta > 0$ es una tasa de aprendizaje inicial.

Si no disponemos de la derivada de la función, podemos aproximarla numéricamente. Cuando realizamos una aproximación numérica de la función, estamos empleando el algoritmo de Kiefer-Wolfowitz⁴⁵⁹:

$$\Delta x = x_{n+1} - x_n = -\eta_n \frac{f(x_n + c_n) - f(x_n - c_n)}{2c_n}$$

donde la secuencia $\{c_n\}$ determina las diferencias finitas utilizadas para aproximar la derivada de una función univariable, originalmente $c_n = n^{-1/3}$.

El método de Kiefer-Wolfowitz fue ampliado posteriormente por Blum para el caso multivariante. En este caso, la estimación del gradiente requiere, al menos, $d + 1$ evaluaciones para una función f con d variables.⁴⁶⁰ En consecuencia, cuando el número de variables es elevado, como sucede en el entrenamiento de redes neuronales artificiales, resulta fundamental poder disponer del gradiente de la función de coste o pérdida que deseamos optimizar.

Si la función que estamos minimizando es doblemente diferenciable y continua, además de fuertemente convexa, el algoritmo de Robbins-Monro alcanza una tasa de convergencia asintóticamente óptima, $E[f(x_n) - f^*] \in O(1/n)$, donde f^* es el mínimo de $f(x)$. En el caso convexo general, sin la suposición de suavidad, también consigue la tasa asintóticamente óptima de convergencia, que, en este caso, es de orden $O(1/\sqrt{n})$.

Aunque, desde un punto de vista teórico, el método de Robbins y Monro puede alcanzar la tasa de convergencia óptima, su algoritmo es demasiado sensible en la práctica a la elección de la secuencia de tamaños de pasos (la estrategia de selección y ajuste de tasas de aprendizaje).

Existen técnicas matemáticas que permiten obtener la tasa de convergencia óptima, como emplear $\eta_n = \nabla^2 f(x^*)^{-1}/n$. No obstante, estas técnicas suelen requerir que dispongamos, a priori, de información a la que normalmente no tenemos acceso.

⁴⁵⁸ Herbert Robbins y Sutton Monroe. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951. ISSN 0003-4851. URL <http://www.jstor.org/stable/2236626>

⁴⁵⁹ Jack Kiefer y Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 09 1952. DOI: 10.1214/aoms/1177729392

⁴⁶⁰ Julius R. Blum. Multidimensional stochastic approximation methods. *The Annals of Mathematical Statistics*, 25(4):737–744, 12 1954. DOI: 10.1214/aoms/1177728659

Por este motivo, Boris T. Polyak^{461,462} en la Unión Soviética (Instituto de Ciencias de Control, Moscú) y David Ruppert⁴⁶³ en Estados Unidos (Universidad de Cornell, Nueva York) desarrollaron de forma independiente un algoritmo que garantiza la convergencia óptima en problemas de optimización convexa. Este algoritmo se basa en suavizar la trayectoria que siguen los valores de los parámetros x , lo que se consigue promediando sus valores x_n . De ahí que el método reciba el nombre de promediado de Polyak o promediado de Polyak-Ruppert.

Este suavizado de la trayectoria puede ser útil, en problemas de optimización, cuando se salta de un lado a otro de un valle alrededor de un mínimo sin visitar en ningún momento un valor cercano al mínimo. Aunque no sabemos la posición exacta del mínimo, que se halla en el fondo del valle, sí que podemos intuir que la media de las posiciones que visitamos en sus laderas puede quedar cerca de ese mínimo.

Aunque Polyak proponía utilizar la media aritmética de todos los valores x_n , en la práctica, para entrenar redes neuronales, se suele emplear una media móvil con un suavizado exponencial:⁴⁶⁴

$$\hat{x}_n = \alpha \hat{x}_{n-1} + (1 - \alpha) x_n$$

donde α es un hiperparámetro más del algoritmo que determina el grado de suavizado. Si usamos $\alpha = 0$, nos quedamos sólo con el último valor, x_n , sin realizar promediado alguno. Para un valor constante, $\alpha < 1$, la contribución de x_{n-k} a la media móvil \hat{x}_n será proporcional a α^k , de ahí lo de suavizado exponencial. Si empleásemos un valor variable $\alpha_n = (n-1)/n$, estaríamos usando una media aritmética de todos los valores de la serie $\{x_n\}$, tal como en la propuesta original de Polyak.

Momentos

Igual que promediar los puntos que vamos visitando a lo largo de un proceso iterativo de optimización puede ayudarnos a localizar el mínimo más rápidamente, idea que dio lugar al promediado de Polyak, también podemos promediar los gradientes observados para intentar acelerar la convergencia de un algoritmo de optimización cuando la función que deseamos minimizar tiene una curvatura pronunciada, gradientes pequeños pero consistentes o sólo disponemos de estimaciones del gradiente con ruido, como sucede al utilizar el gradiente descendente estocástico. El método resultante se denomina uso de momentos, partiendo de una analogía física no demasiado acertada. Curiosamente, fue propuesto por Boris Polyak⁴⁶⁵ y se viene utilizando desde que se empezaron a entrenar redes neuronales multicapa usando *backpropagation* y el gradiente descendente.⁴⁶⁶

El uso de momentos se basa en introducir una variable v que representa la velocidad a la que se mueven los parámetros x de la función $f(x)$ que

⁴⁶¹ Boris T. Polyak. New method of stochastic approximation. *Automation and Remote Control (translated from Russian)*, 51(7.2):937–946, 1990. URL <https://goo.gl/CzEdSH>

⁴⁶² Boris T. Polyak y A. B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, July 1992. ISSN 0363-0129. DOI: 10.1137/0330046

⁴⁶³ David Ruppert. Efficient estimators from a slowly convergent Robbins-Monro process. Technical Report 781, School of Operations Research and Industrial Engineering, Cornell University, 1988. URL <https://goo.gl/VdvSzr>

⁴⁶⁴ Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, y Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *arXiv e-prints*, arXiv:1512.00567, 2015b. URL <http://arxiv.org/abs/1512.00567>

⁴⁶⁵ Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1 – 17, 1964. ISSN 0041-5553. DOI: 10.1016/0041-5553(64)90137-5

⁴⁶⁶ David E. Rumelhart, Geoffrey E. Hinton, y Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, October 1986a. DOI: 10.1038/323533a0

deseamos minimizar durante un proceso iterativo de optimización. Igual que en el promediado de Polyak realizábamos un suavizado exponencial de los puntos \hat{x}_n en los que se evaluaba la función, al utilizar momentos realizaremos un suavizado exponencial de las velocidades \hat{v}_n con las que van moviéndose los puntos x_n .

El gradiente negativo utilizado por el gradiente descendente se interpreta como una fuerza que mueve una partícula en el espacio de los parámetros de la función. En mecánica clásica, se define la cantidad de movimiento o momento lineal [*momentum*] como el producto de la masa de un cuerpo y su velocidad en un momento determinado: $p = m v$.

Cuando dos objetos se mueven a la misma velocidad, p.ej. una pelota de tenis o una bola de demolición, el de menor masa es mucho más fácil de detener que el de mayor masa (a nadie se le ocurriría intentar detener con la mano una bola de demolición, algo que sí podemos hacer fácilmente con una pelota de tenis). De ahí proviene la idea de cantidad de movimiento.

En un algoritmo de optimización basado en el gradiente descendente, no tenemos partículas con diferentes masas, por lo que podemos asumir una masa unitaria y que el vector v asociado a la velocidad representa la cantidad de movimiento p de la partícula, su momento.

Un hiperparámetro α se utiliza para calcular una media móvil de los gradientes, usando un suavizado exponencial, y el resultado es el que se utiliza para actualizar los valores de los parámetros en nuestro problema de optimización:

$$\begin{aligned} v_n &= \alpha v_{n-1} - \eta \nabla f(x_n) \\ \Delta x &= x_{n+1} - x_n = v_n \end{aligned}$$

Cuanto mayor sea el hiperparámetro α con respecto a la tasa de aprendizaje, mayor será la contribución de los gradientes evaluados previamente a la dirección en la que se actualiza x .

El tamaño del cambio Δx depende de cómo de pronunciados sean los gradientes y de cómo de alineadas se encuentren las estimaciones sucesivas del gradiente utilizadas por el gradiente descendente estocástico. Al incorporar momentos, Δx irá aumentando en la dirección en la que se encuentren gradientes consistentes.

El cambio Δx será máximo cuando muchos gradientes sucesivos apunten siempre en la misma dirección. En el caso extremo, si observamos siempre el mismo gradiente g , el movimiento acelerará en la dirección $-g$ hasta alcanzar una velocidad terminal

$$v_{max} = \frac{\eta \|g\|}{1 - \alpha}$$

Esta interpretación nos puede servir para darle un sentido intuitivo al valor que elijamos para el hiperparámetro α . Un valor habitual, $\alpha = 0.9$, equivale a multiplicar por 10 la velocidad máxima que se puede conseguir al utilizar el gradiente descendente. El factor $1/(1 - \alpha)$ determina esa proporción y nos da una idea de la forma en la que el uso de momentos acelera la convergencia del gradiente descendente. Si empleásemos $\alpha = 0.5$, como mucho nos podríamos mover dos veces más rápido que con el gradiente descendente sin momentos.

Igual que el promediado de Polyak, el uso de momentos pretende aminorar los problemas que puede ocasionar la varianza en la estimación estocástica del gradiente en las técnicas iterativas de optimización basadas en el gradiente descendente.

Como sucede con la tasa de aprendizaje, es habitual utilizar un valor variable para el hiperparámetro α . Normalmente, se emplea un valor menor durante las primeras iteraciones del algoritmo, en las que preferimos guiarnos por la señal recibida a través del gradiente. Después, se suele incrementar el valor de α , lo que puede ayudar si nos movemos en zonas planas de la superficie de la función que deseamos minimizar, en las que el gradiente es prácticamente cero. Es decir, justo al revés de lo que haríamos con la tasa de aprendizaje, que comienza con un valor elevado que vamos reduciendo conforme avanza la ejecución del algoritmo. En el caso del momento, una estrategia habitual consiste en comenzar con un momento $\alpha = 0.5$ e ir incrementándolo paulatinamente hasta llegar a $\alpha = 0.99$.

Momentos tradicionales

Normalmente, en redes neuronales, el hiperparámetro α se representa por la letra griega μ . Para enrevesar un poco más la analogía física, el parámetro μ se denomina momento. Así pues, con el término momento se hace referencia tanto al método en sí, que considera la velocidad del movimiento, como al hiperparámetro utilizado en el suavizado exponencial del gradiente.

Si expresamos la fórmula de actualización de los parámetros de la función en los términos habituales que se emplean para entrenar redes neuronales, la regla de actualización de los pesos que se utiliza cuando empleamos momentos es de la forma:

$$\Delta w_n = -\eta \nabla f(w_n) + \mu \Delta w_{n-1}$$

donde w_n es el valor del vector de pesos de la red en la n -ésima iteración del algoritmo y $\nabla f(w_n)$ es la evaluación del gradiente del error de la función f de error, coste o pérdida para el valor actual de los parámetros de la red, w_n .

Hemos visto que el parámetro μ se denomina, incorrectamente, momento. Si realizamos una interpretación física más intuitiva de la fórmula

anterior, en la que podemos ver Δw_n como una velocidad, podríamos considerar que w_n representa la posición de una partícula sometida a dos fuerzas. La primera de ellas, proporcional al gradiente descendente, corresponde a la aceleración que sufre la partícula al desplazarse sobre una superficie inclinada. La segunda de ellas, asociada al momento, representa más bien una inercia: la tendencia natural a continuar moviéndose en la misma dirección en la que venía haciéndolo. En este caso, la inercia está sujeta a cierto grado de fricción, $1 - \mu$, que hará que la partícula reduzca paulatinamente su velocidad hasta acabar deteniéndose en ausencia de gradiente.

El momento, o inercia si lo prefiere, ha de ser lo suficientemente débil para permitir que el gradiente guíe el movimiento hasta un mínimo de la función, donde nos detendremos. Al mismo tiempo, ha de ser lo suficientemente fuerte para prevenir que el movimiento se detenga demasiado al atravesar una zona relativamente plana de la función. El uso de momentos implica la existencia de una fricción viscosa, proporcional a la velocidad de la partícula, $-v$. En la Naturaleza, la fricción también puede ser turbulenta (proporcional al cuadrado de la velocidad, como la experimentada por un avión) o seca (constante, como una masa cayendo por un plano inclinado). En el primer caso, la fricción $-v^2$ es demasiado pequeña cuando la velocidad es reducida, lo que nos impediría detenernos al llegar a un mínimo. En el segundo, la fricción constante puede ser excesiva, con lo que nos detendríamos antes de llegar al mínimo. Una fricción viscosa se halla en un punto intermedio que nos permite conseguir el efecto deseado, además de resultar conveniente desde el punto de vista computacional.

Si interpretamos el uso de momentos como la introducción de una fuerza que causa una aceleración en el movimiento de una partícula y recordamos que la Segunda Ley de Newton relaciona fuerzas con aceleraciones ($F = ma$), podemos ver que, en realidad, cuando utilizamos momentos, estamos recurriendo a información relacionada con la segunda derivada de la función que pretendemos minimizar: la aceleración ocasionada por una fuerza equivale a la derivada de la velocidad, que es la segunda derivada de la posición de la partícula. En otras palabras, desde el punto de vista formal, al usar momentos es como si añadiésemos a la regla de actualización de los pesos información acerca de la segunda derivada de la función que pretendemos minimizar.

La segunda derivada de la función, en forma de matriz hessiana para el caso multidimensional, nos proporciona información acerca de cómo varía ese gradiente (la curvatura de la función). El uso de momentos nos permite aprovechar parte de la información que nos podría proporcionar esa segunda derivada sin necesidad de calcular la matriz hessiana, algo que haremos de forma explícita cuando recurramos a técnicas de optimización de segundo orden.

En realidad, la matriz hessiana puede ser enorme si el número de variables es elevado, por lo que nos tendremos que conformar con las aproximaciones de la matriz hessiana que ofrecen técnicas de optimización como L-BFGS.

Definiendo de forma explícita la velocidad v del cambio al que sometemos los parámetros de la red, Δw , el uso tradicional de momentos en el entrenamiento de una red neuronal se puede expresar de la siguiente forma:

$$\begin{aligned} v_n &= \mu v_{n-1} - \eta \nabla f(w_n) \\ \Delta w &= w_{n+1} - w_n = v_n \end{aligned}$$

Esta forma de representar la actualización de los pesos nos ayudará a comprender mejor una variante del uso de momentos: los momentos de Nesterov.

Momentos de Nesterov

En la versión tradicional de los momentos, se evalúa el gradiente del error en la posición actual y , además del movimiento asociado a ese gradiente, se da un salto en función del gradiente observado previamente. Una versión mejorada, basada en el método de optimización de funciones convexas de Yurii Nesterov,⁴⁶⁷ primero da un salto en la dirección del gradiente previamente observado y luego evalúa el gradiente del error, en la posición a la que se llega tras realizar ese salto:

$$\begin{aligned} v_n &= \mu v_{n-1} - \eta \nabla f(w_n + \mu v_{n-1}) \\ \Delta w &= w_{n+1} - w_n = v_n \end{aligned}$$

La evaluación del gradiente, en este caso, se realiza tras el salto en la dirección que nos marca el momento. El momento de Nesterov, pues, se basa en corregir un error después de cometerlo, lo que parece tener más sentido desde el punto de vista intuitivo y contribuye a acelerar la convergencia del algoritmo de optimización basado en momentos.⁴⁶⁸

Desde el punto de vista formal, los momentos de Nesterov ofrecen una mejor tasa de convergencia en la optimización de funciones convexas, que pasa de ser $O(1/n)$ a $O(1/n^2)$. En la práctica, para entrenar redes neuronales artificiales, los momentos de Nesterov funcionan ligeramente mejor que los momentos estándar, aunque sin proporcionar mejoras asintóticas en la tasa de convergencia del gradiente descendente estocástico.

En vez de utilizar las expresiones anteriores para el uso de momentos de Nesterov, se puede reorganizar la forma de expresar la actualización de los pesos de forma que la expresión resultante se parezca más a la utilizada por el gradiente descendente estocástico. Ya que el momento de Nesterov se basa en anticipar la posición de una partícula debida a la inercia de su movimiento, $w_{ahead} = w + \mu v$, podemos manipular las expresiones anteriores para almacenar siempre w_{ahead} en vez de w :

⁴⁶⁷ Yurii Nesterov. A method of solving a convex programming problem with convergence rate $O(1/\text{sqr}(k))$. *Soviet Mathematics Doklady*, 27:372–376, 1983

⁴⁶⁸ Ilya Sutskever, James Martens, George Dahl, y Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. En Sanjoy Dasgupta y David McAllester, editores, *ICML'13 Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/sutskever13.pdf>

$$\begin{aligned}v_n &= \mu v_{n-1} - \eta \nabla f(w_n) \\ \Delta w &= w_{n+1} - w_n = -\mu v_{n-1} + (1 + \mu)v_n\end{aligned}$$

La única diferencia entre los momentos de Nesterov y los momentos estándar es el punto en el que se evalúa el gradiente. Al usar momentos de Nesterov, se estima el gradiente tras aplicar el movimiento debido a la inercia acumulada. Se puede ver como una forma de añadir un factor de corrección al uso tradicional de momentos en los algoritmos de optimización basados en el gradiente descendente estocástico.

De hecho, es habitual que las bibliotecas utilizadas en *deep learning* incorporen momentos en los algoritmos de optimización que ofrecen a sus usuarios. Por ejemplo, en el caso de Keras, <https://keras.io/optimizers/>, su optimizador SGD basado en el gradiente descendente estocástico admite los siguientes parámetros: una tasa de aprendizaje inicial `lr`, un factor `decay` de reducción de la tasa de aprendizaje, un momento constante `momentum` y una variable booleana que nos permite elegir entre momentos estándar y momentos de Nesterov (`nesterov`), además de dos parámetros para recortar el gradiente (tanto su valor, `clipvalue`, elemento a elemento, como su norma, `clipnorm`), que son comunes a todos los algoritmos de optimización ofrecidos por la herramienta.

Tasas de aprendizaje adaptativas

En la práctica, se hace necesario disminuir gradualmente la tasa de aprendizaje utilizada por el gradiente descendente estocástico. Es decir, en vez de utilizar una tasa fija de aprendizaje, η , se emplea una tasa variable, $\eta(t)$, que va reduciéndose conforme avanza el proceso de optimización basado en el gradiente descendente:

$$\Delta x(t) = -\eta(t) \nabla f(x(t))$$

¿Por qué resulta necesario ir reduciendo la tasa de aprendizaje? Básicamente, porque la estimación utilizada del gradiente, $\nabla f(x(t))$, incluye una fuente de error debida al muestreo de los ejemplos del conjunto de entrenamiento. Ese error no disminuye cuando nos acercamos a un mínimo de la función de coste, que es cuando nos gustaría realizar movimientos más precisos. El gradiente real se acercará a cero cuando estemos cerca de un mínimo, no así su estimación estocástica. De ahí que resulte mucho más importante utilizar tasas de aprendizaje adaptativas cuando se utiliza el gradiente descendente estocástico (cuando utilicemos aprendizaje por lotes, la estimación del gradiente será más precisa y tal vez nos podamos apañar con una tasa de aprendizaje constante).

En esta sección, utilizaremos la variable t para indicar la iteración en lugar de n , con el objetivo de que se pueda distinguir mejor visualmente de la letra griega eta, η , que usamos para denotar la tasa de aprendizaje.

Existen múltiples estrategias para ir adaptando las tasas de aprendizaje, que en ocasiones se denominan planes o esquemas de enfriamiento [*annealing schedules*], por analogía con el ajuste de temperatura que se utiliza en las técnicas de enfriamiento simulado [*simulated annealing*], una metaheurística para resolver problemas de optimización inspirado en el proceso de fabricación de aleaciones metálicas como el acero o de materiales cerámicos. Esos materiales se fabrican calentando el material hasta fundirlo y controlando cuidadosamente su proceso de enfriamiento, que se realiza lentamente para que el material alcance las propiedades físicas deseadas. Dicho enfriamiento les permite recristalizar en configuraciones con un menor nivel de energía (un mínimo “global” de la función de energía) que en su estado inicial (un mínimo local de energía).

Algunas de las estrategias comunes para ir variando la tasa de aprendizaje $\eta(t)$ son las siguientes:

- *Disminución lineal hasta la iteración k:*

$$\eta(t) = \left(1 - \frac{t}{k}\right) \eta_0 + \eta_k$$

A partir de la iteración k , la tasa de aprendizaje se mantiene constante, η_k . Por ejemplo, se puede escoger de forma heurística el valor de η_k para que corresponda al 1 % del valor de la tasa de aprendizaje inicial η_0 ; esto es, $\eta_k = 0.01\eta_0$.

- *Factor de reducción δ:*

$$\eta(t) = \frac{1}{1 + \delta t} \eta_0$$

donde la tasa de aprendizaje se mantiene constante cuando $\delta = 0$ y se va reduciendo paulatinamente cuando $\delta > 0$. Al comienzo, la reducción se va realizando lentamente si usamos valores pequeños de δ (p.ej. $\delta < 0.01$). Cuando $\delta t > 1$, la tasa de aprendizaje pasa a decrecer linealmente. Es la estrategia de búsqueda y convergencia⁴⁶⁹ propuesta originalmente por Christian Darken y John Moody. Se utiliza en las implementaciones del gradiente descendente estocástico de herramientas como Keras.

- *El “conductor audaz” [bold driver]:*^{470,471}

$$\eta(t) = \begin{cases} \rho^+ \eta(t-1) & \text{si } \Delta f(x) < 0 \\ \rho^- \eta(t-1) & \text{si } \Delta f(x) > 0 \end{cases}$$

donde la tasa de aprendizaje se ajusta dinámicamente monitorizando las variaciones de la función que pretendemos minimizar, siendo ρ^+ una constante ligeramente mayor que 1 (típicamente, 1.1) y ρ^- una constante significativamente menor que 1 (típicamente, 0.5).

⁴⁶⁹ Christian Darken y John Moody. Note on learning rate schedules for stochastic optimization. En *Proceedings of the 3rd International Conference on Advances in Neural Information Processing Systems*, NIPS'90, pages 832–838. Morgan Kaufmann Publishers Inc., 1990. ISBN 1-55860-184-8. URL <https://goo.gl/HuCSXu>

⁴⁷⁰ Roberto Battiti. Accelerated Backpropagation Learning: Two Optimization Methods. *Complex Systems*, 3(4):331–342, 1980. ISSN 0891-2513. URL http://www.complex-systems.com/abstracts/v03_i04_a02.html

⁴⁷¹ T. P. Vogl, J. K. Mangis, A. K. Ringer, W. T. Zink, y D. L. Alkon. Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, 59(4):257–263, Sep 1988. ISSN 1432-0770. doi: 10.1007/BF00332914

- *Tasas locales de aprendizaje*, con tasas adaptables que se ajustan individualmente para cada parámetro de la función $f(x)$, p.ej.

$$\eta_i(t) = \eta g_i(t)$$

donde η es la tasa de aprendizaje inicial (la misma para todos los parámetros) y g_i es un factor de ganancia local (initialmente 1), el cual se incrementa cuando el gradiente para un peso no cambia de signo y se disminuye cuando lo hace, para lo que se comprueba el signo del producto de dos estimaciones consecutivas del gradiente para el parámetro x_i : $\delta_i(t) \delta_i(t-1)$.

$$g_i(t) = \begin{cases} g_i(t-1) + \delta & \text{si } \delta_i(t) \delta_i(t-1) > 0 \\ g_i(t-1) * (1 - \delta) & \text{si } \delta_i(t) \delta_i(t-1) < 0 \end{cases}$$

El método, propuesto por Hinton en su curso MOOC de Coursera, realiza aumentos aditivos y descensos multiplicativos en los factores de ganancia, como en la regla delta-barra-delta,⁴⁷² usando un único parámetro $\delta = 0.05$. De esta forma, las ganancias elevadas se reducen exponencialmente cuando se producen oscilaciones. Si el gradiente es totalmente aleatorio, las ganancias se mantienen en torno a 1 (sumamos $+\delta$ la mitad de las veces y multiplicamos por $1 - \delta$ la otra mitad). Esto favorece la estabilidad del algoritmo de ajuste de las tasas de aprendizaje.

Como sucede con el gradiente, los factores de ganancia también se pueden recortar, para asegurarnos de que se mantienen en un rango razonable, p.ej. [0.1, 10] si queremos ser conservadores o [0.01, 100] si somos algo más atrevidos.

- *Quickprop*⁴⁷³ cambia la forma de actualizar los parámetros x de la función $f(x)$. Quickprop intenta evitar fluctuaciones bruscas (como el recorte del gradiente) y emplea dos evaluaciones consecutivas del gradiente como aproximación discreta de la segunda derivada de la función, lo que permite acelerar la convergencia del algoritmo ajustando dinámicamente los valores de cada parámetro x_i :

$$\Delta x_i(t) = \begin{cases} \alpha_i(t) \Delta x_i(t-1) & \text{si } \Delta x_i(t-1) \neq 0 \\ \eta_0 \delta_i(t) & \text{si } \Delta x_i(t-1) = 0 \end{cases}$$

donde

$$\alpha_i(t) = \min \left\{ \frac{\delta_i(t)}{\delta_i(t-1) - \delta_i(t)}, \alpha_{max} \right\}$$

El parámetro α_{max} se utiliza para acotar el tamaño de las actualizaciones (típicamente, $\alpha_{max} = 1.75$). η_0 es la tasa de aprendizaje inicial, que sólo se utiliza al comienzo del algoritmo ($0.01 \leq \eta_0 \leq 0.6$). Como antes, $\delta_i(t)$ representa la estimación del gradiente realizada para el parámetro x_i durante la iteración t .

⁴⁷² Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295 – 307, 1988. ISSN 0893-6080. DOI: 10.1016/0893-6080(88)90003-2

⁴⁷³ Scott E. Fahlman. Faster-Learning Variations on Back-Propagation: An Empirical Study. En David S. Touretzky, Geoffrey E. Hinton, y Terrence J. Sejnowski, editores, *Proceedings of the 1988 Connectionist Models Summer School*, pages 38–51. Morgan Kaufmann, 1988. URL <https://www.cs.cmu.edu/~sef/sefPubs.htm>

En realidad, el algoritmo Quickprop es un método quasi-Newton basado en el algoritmo de la secante. Los métodos quasi-Newton son técnicas de optimización de segundo orden derivados del método de Newton, con el que nos encontraremos más adelante. Como su nombre indica, se basan en utilizar la segunda derivada de la función que pretendemos minimizar.

Los anteriores son sólo algunos de los mecanismos de ajuste automático de las tasas de aprendizaje que se han propuesto para el entrenamiento de redes neuronales. Cada uno de ellos incluye sus propios hiperparámetros, que debemos ajustar. La elección de valores adecuados para estos hiperparámetros puede resultar crítica para el entrenamiento con éxito de una red neuronal. A menudo, no obstante, se deja a discreción de la herramienta que estemos utilizando, puede que con los valores por defecto sugeridos por sus autores. Sin embargo, deberíamos ajustarlos como cualquier otro hiperparámetro del algoritmo de entrenamiento de la red. Podemos recurrir a una simple estrategia de prueba y error, en la que monitorizaremos las curvas de aprendizaje que muestran el valor de la función objetivo conforme avanza el proceso iterativo de optimización (evaluada sobre un conjunto de validación independiente del conjunto de entrenamiento, claro está). O, mejor aún, recurrir a alguna técnica automática de ajuste de los hiperparámetros del entrenamiento de la red.

Dada la importancia clave que tiene la tasa de aprendizaje en los resultados que se obtienen al aplicar un algoritmo de optimización para entrenar una red neuronal artificial, no debería sorprendernos demasiado que se hayan propuesto numerosos métodos para ajustar automáticamente las tasas de aprendizaje que gobiernan el proceso de aprendizaje de una red neuronal. Es más que probable que nos encontremos con alguno de ellos si recurrimos a alguna herramienta de *deep learning*, las cuales proporcionan, ya implementados, diferentes algoritmos de entrenamiento de redes neuronales (Torch, Theano, TensorFlow, Keras, CNTK [*Microsoft Cognitive Toolkit*], MXNet, DeepLearning4j o Caffe son algunas de las más populares).

Veamos, a continuación, algunos de los métodos que se han propuesto recientemente:

AdaGrad

Una posibilidad consiste en ajustar individualmente las tasas de aprendizaje para cada parámetro de la red haciendo que el ajuste sea inversamente proporcional a la raíz cuadrada de la suma de los valores históricos de los gradientes al cuadrado. Es la estrategia empleada por AdaGrad, cuyo nombre proviene del uso de “gradientes adaptativos”.⁴⁷⁴

El método AdaGrad va acumulando la suma de los cuadrados de los gradientes de la función de error, coste o pérdida para cada parámetro

⁴⁷⁴ John Duchi, Elad Hazan, y Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, July 2011. ISSN 1532-4435. URL <http://jmlr.org/papers/v12/duchi11a.html>

de la red:

$$r(t) = r(t-1) + g(t) \odot g(t)$$

donde $g(t)$ es el vector del gradiente estimado en la iteración t del algoritmo; esto es, la estimación con la que trabajamos de $\nabla f(x(t))$.

La tasa de aprendizaje que utiliza AdaGrad viene dada por

$$\eta_i(t) = \frac{\eta}{\sqrt{r_i(t)} + \epsilon} g_i(t)$$

siendo η una tasa de aprendizaje general (podemos verla como la tasa de aprendizaje inicial, usualmente $\eta = 0.01$) y ϵ una constante pequeña que se utiliza para evitar divisiones por cero (p.ej. del orden de 10^{-4}). Curiosamente, si no utilizásemos la raíz cuadrada, el algoritmo funcionaría mucho peor.

Usando notación vectorial, la expresión anterior da lugar a la siguiente fórmula de actualización de los parámetros de la red

$$\Delta x(t) = -\frac{\eta}{\sqrt{r(t)} + \epsilon} \odot g(t)$$

Los parámetros de la red para los que se observa una mayor derivada parcial de la función de error o pérdida son los que más ven reducidas sus tasas de aprendizaje. En el extremo opuesto, los parámetros para los que normalmente apenas se observa un gradiente, conservan sin reducción sus tasas locales de aprendizaje. De esta forma, los parámetros que se modifican con frecuencia actualizan su valor poco a poco, mientras que los parámetros que sólo se modifican infrecuentemente cambian más rápidamente de valor. El efecto neto de AdaGrad es conseguir que se avance más rápidamente en las direcciones en las que la superficie de error es más plana.

Aunque AdaGrad puede poseer algunas propiedades deseables desde un punto de vista teórico, la acumulación de gradientes al cuadrado en el denominador puede ocasionar que las tasas de aprendizaje efectivas disminuyan demasiado de forma prematura. Cuando son demasiado pequeñas, llega un momento en el que la red es incapaz de aprender nada más.

AdaDelta

AdaDelta⁴⁷⁵ es una extensión de AdaGrad diseñada para aminorar la reducción agresiva de las tasas de aprendizaje empleada por AdaGrad. Para ello, en vez de acumular todos los gradientes observados en el pasado, se limita a una ventana de ancho limitado w . Es decir, se basa en la idea de considerar únicamente los últimos w gradientes observados a la hora de ajustar las tasas de aprendizaje.

Sin embargo, almacenar los w últimos gradientes puede resultar demasiado ineficiente, motivo por el que AdaDelta propone utilizar una media

⁴⁷⁵ Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv e-prints*, arXiv:1212.5701, 2012. URL <http://arxiv.org/abs/1212.5701>

móvil con suavizado exponencial, como en el promediado de Polyak o en el uso de momentos:

$$E[g^2](t) = \gamma E[g^2](t-1) + (1-\gamma)g^2(t)$$

con γ una constante alrededor de 0.9.

La expresión que utilizábamos en AdaGrad se puede considerar como una forma de emplear el error cuadrático medio:

$$\Delta x(t) = -\frac{\eta}{E[g^2](t) + \epsilon} \odot g(t) = -\frac{\eta}{RMS[g](t)} \odot g(t)$$

Las unidades en esta fórmula no coinciden, igual que tampoco lo hacen en el gradiente descendente estocástico o al utilizar momentos, ya que ambos lados deberían ir expresados en las unidades (hipotéticas) de los parámetros de la red. Si tenemos en cuenta que el gradiente g se mide en las mismas unidades que Δx , de ahí podemos concluir que, al estar dividiendo $RMS[g]$, también expresado en las mismas unidades que el gradiente g , debemos multiplicar por algo que se exprese en las mismas unidades Δx .

Por este motivo, Matthew Zeiler, al proponer AdaDelta, sugirió emplear otra media móvil con suavizado exponencial, esta vez definida no sobre los cuadrados de los gradientes, sino sobre los cuadrados de las actualizaciones de los parámetros:

$$E[\Delta x^2](t) = \gamma E[\Delta x^2](t-1) + (1-\gamma)\Delta x^2(t)$$

El error cuadrático medio de las actualizaciones de los parámetros se puede aproximar a partir de esa media móvil:

$$RMS[\Delta x](t) = \sqrt{E[\Delta x^2]} + \epsilon$$

con ϵ añadido, simplemente, por mantener la simetría con la expresión utilizada por AdaGrad.

De esta forma, llegamos a la fórmula final de actualización de los parámetros propuesta por AdaDelta:

$$\Delta x(t) = -\frac{RMS[\Delta x](t-1)}{RMS[g](t)} \odot g(t)$$

Observe que ni siquiera necesitamos definir una tasa de aprendizaje por defecto. Hemos conseguido una estrategia de ajuste dinámico de las tasas de aprendizaje que no emplea ningún hiperparámetro.

Al usar AdaDelta, las tasas de aprendizaje efectivas son mayores en las primeras capas de la red y menores para las capas finales, lo que ayuda a compensar el hecho de que los gradientes disminuyan conforme nos alejamos de la capa de salida de la red.

A diferencia de otras técnicas de ajuste adaptativo de las tasas de aprendizaje, en las etapas finales del entrenamiento (cuando los gradientes observados son muy pequeñas), la tasa de aprendizaje efectiva tiende a uno (debido al uso de las constantes ϵ). Algo que podría ocasionar severos problemas de convergencia en otros métodos de optimización, no supone impedimento alguno para AdaDelta, ya que las actualizaciones de los parámetros tenderán a cero (debido al factor del gradiente, ya cercano a cero). Eso sí, si combinásemos AdaDelta con el uso de momentos, sí se podrían producir oscilaciones no deseadas: los momentos, que en otros métodos ayudan a reducir oscilaciones, en AdaDelta se acumulan en el numerador. Así pues, no utilice momentos si opta por seleccionar AdaDelta como método de optimización.

En líneas generales, todo parece indicar que AdaDelta puede converger más rápidamente que AdaGrad y, además, es capaz de obtener mejores soluciones, ya que no se queda atascado hasta el punto de ser incapaz de aprender nada nuevo, como sí sucede con AdaGrad. Pese a ello, AdaGrad es más popular en la práctica.

Rprop

Ya en los años 90, Martin Riedmiller y Heinrich Brau, de la Universidad de Karlsruhe en Alemania, propusieron una variante resiliente de *backpropagation* a la que denominaron Rprop [*Resilient backPropagation*].⁴⁷⁶

Para conseguir una versión resiliente de *backpropagation*, Rprop prescinde de la magnitud del gradiente a la hora de determinar el tamaño del salto que se realiza al actualizar los parámetros. Es decir, sólo se considera el signo del gradiente. La actualización de los parámetros de la red se efectúa de acuerdo a la secuencia de los signos de las derivadas parciales de la función de coste o pérdida con respecto a cada uno de los parámetros.

Igual que sucede con otras heurísticas de ajuste adaptativo de las tasas de aprendizaje, se comparan los signos de las dos últimas estimaciones del gradiente: $C_i(t) = g_i(t-1)g_i(t)$.

- Si el gradiente no cambia de signo, $C_i(t) > 0$, damos un salto en la dirección del gradiente descendente, de una magnitud $\Delta_i(t)$ ligeramente superior al último salto $\Delta_i(t-1)$ que dimos en esa dirección, usando una constante $\eta_0^+ > 1$:

$$\Delta_i(t) = \min\{\eta_0^+ \Delta_i(t-1), \Delta_{max}\}$$

$$\Delta x_i(t) = -\text{sign}(g_i(t)) \cdot \Delta_i(t)$$

- Si el gradiente cambia de signo, $C_i(t) < 0$, revertimos el último cambio que hicimos:

$$\Delta_i(t) = \max\{\eta_0^- \Delta_i(t-1), \Delta_{min}\}$$

⁴⁷⁶ Martin Riedmiller y Heinrich Braun. A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. En *IEEE International Conference on Neural Networks*, pages 586–591 vol.1, 1993. DOI: 10.1109/ICNN.1993.298623

La RAE define la resiliencia como la capacidad de adaptación de un ser vivo frente a un agente perturbador o un estado o situación adversos. Su segunda acepción hace referencia a la capacidad de un material, mecanismo o sistema para recuperar su estado inicial cuando ha cesado la perturbación a la que ha estado sometido.

$$\Delta x_i(t) = -\Delta x_i(t-1)$$

Además, nos preparamos para que el siguiente salto que demos sea de menor magnitud que el anterior ($\eta_0^- < 1$) y, para evitar una doble penalización en la siguiente iteración, en la que el signo del gradiente esperamos que vuelva a cambiar, no adaptaremos de nuevo la magnitud del salto y fijaremos $g_i(t) = 0$ para tratar esta situación de forma especial:

- Si $C_i(t) = 0$, al comienzo de la ejecución del algoritmo o tras un cambio en el signo del gradiente:

$$\Delta_i(t) = \Delta_i(t-1)$$

$$\Delta x_i(t) = -\text{sign}(g_i(t)) \cdot \Delta_i(t)$$

Siempre damos un salto en dirección del gradiente descendente, salvo cuando detectamos un cambio en el signo del gradiente, momento en el que retrocedemos a la posición previa del parámetro para el que cambia el signo del gradiente.

La magnitud del cambio al que someteremos a cada parámetro de la red se irá adaptando en función de los parámetros $0 < \eta_0^- < 1 < \eta_0^+$. Típicamente, $\eta_0^+ = 1.2$ y $\eta_0^- = 0.5$. Cuando el gradiente no cambia de signo, iremos ampliando el tamaño del salto: $\eta_0^+ \Delta_i(t-1)$ con $\eta_0^+ > 1$. Cuando el gradiente cambia de signo, retrocederemos y volveremos a intentarlo en la siguiente iteración, pero dando un salto más pequeño: $\eta_0^- \Delta_i(t-1)$, con $\eta_0^- < 1$.

El valor inicial $\Delta_i(0)$ no es demasiado crítico para el funcionamiento del algoritmo y basta con elegir una constante positiva Δ_0 . Por último, las cotas Δ_{min} y Δ_{max} limitan la magnitud de los cambios para evitar errores de tipo numérico (de *underflow* y *overflow*, respectivamente). Por ejemplo, Michael Schuster recomienda utilizar $\Delta_{min} = 10^{-6}$ y $\Delta_{max} = 50$ en su tesis doctoral.⁴⁷⁷

El uso de Rprop acelera la convergencia del gradiente descendente y se propuso para su uso en aprendizaje por lotes. En el gradiente estocástico, los errores debidos al ruido en las estimaciones del gradiente pueden ocasionar cambios de signo espurios en el gradiente observado. En comparación con el gradiente descendente, Quickprop o SuperSAB, el entrenamiento de redes neuronales con Rprop requiere menos épocas de entrenamiento. De hecho, se ha observado experimentalmente que su rendimiento es comparable al de otros métodos más sofisticados, como los basados en el uso de gradientes conjugados.⁴⁷⁸ Estas características lo convierten en uno de los mejores métodos de optimización basados exclusivamente en el uso del gradiente. Además, es robusto frente a problemas de índole numérica, ya que las cotas Δ_{min} y Δ_{max} consiguen un efecto similar al del recorte del gradiente.

Se puede diseñar una variante de Rprop en la que no se realiza el retroceso tras el cambio de signo, lo que simplifica ligeramente la implementación del algoritmo. Esta variante se suele denominar Rprop⁻.

Martin Riedmiller. Advanced Supervised Learning in Multi-layer Perceptrons: From Backpropagation to Adaptive Learning Algorithms. *Computer Standards & Interfaces*, 16(3):265 – 278, 1994. ISSN 0920-5489. DOI: 10.1016/0920-5489(94)90017-5

⁴⁷⁷ Michael Schuster. *On Supervised Learning from Sequential Data with Applications for Speech Recognition*. PhD thesis, Nara Institute of Science and Technology, 1999

⁴⁷⁸ J. M. Hannan y J. M. Bishop. A comparison of fast training algorithms over two real problems. En *Fifth International Conference on Artificial Neural Networks (Conf. Publ. No. 440)*, pages 1–6, Jul 1997. DOI: 10.1049/cp:19970692

El éxito de Rprop causó la aparición de múltiples variantes de este método adaptativo de optimización:

- SASS⁴⁷⁹ utiliza las mismas reglas de actualización de los parámetros que Rprop, pero cambia la forma de calcular la magnitud de los saltos $\Delta_i(t)$. En lugar de utilizar el cambio de signo de los dos últimos gradientes como guía, $g_i(t-1)g_i(t)$, se basa en el método de la bisección para la minimización de una función unidimensional y compara los signos de los tres últimos gradientes:

$$\Delta_i(t) = \begin{cases} 2.0 \Delta_i(t-1) & \text{si } g_i(t)g_i(t-1) \geq 0 \\ & \quad \text{y } g_i(t)g_i(t-2) \geq 0 \\ 0.5 \Delta_i(t-1) & \text{en otro caso} \end{cases}$$

- QRprop [*Quadratic Rprop*] y DERprop [*Diagonal Estimation Rprop*]⁴⁸⁰ son dos híbridos similares que combinan RProp con métodos de optimización de segundo orden, los cuales recurren a la segunda derivada de la función. Tanto QRprop como DERprop cambian de un método a otro, pasando de la estrategia de Rprop a una aproximación de segundo orden, cuando llegan al vecindario de un mínimo local. En el caso de QRprop, se utiliza el método de la secante para el caso unidimensional, mientras que DERprop calcula los elementos de la diagonal de la matriz hessiana.
- Si añadimos la metaheurística del enfriamiento simulado al algoritmo Rprop obtenemos SARprop [*Simulated Annealing Rprop*].⁴⁸¹ El uso de enfriamiento simulado añade ruido a Rprop, con lo que se puede conseguir que Rprop pueda escapar de mínimos locales. El ruido se añade modificando la estimación del gradiente:

$$g_i^{SARprop}(t) = g_i(t) - 0.01 \frac{x_i}{1 + x_i^2} SA$$

donde $SA = 2^{-Tt}$ y T es el parámetro que representa la temperatura en el enfriamiento simulado. El efecto conseguido es similar a utilizar un mecanismo de regularización como *weight decay*. Al comienzo, se obliga a que los pesos sean pequeños, como en *weight decay*. Conforme el entrenamiento de la red progresiona, la penalización se reduce y se permite que los pesos crezcan lo que resulte necesario.

- iRprop [*improved Rprop*] es una versión mejorada de Rprop, reinventada por Christian Igel y Michael Hüskens, de la que existen versiones con y sin retroceso cuando se detecta un cambio de signo en el gradiente: iRprop⁺ (con retroceso) e iRprop⁻ (sin retroceso).⁴⁸²

En Rprop, un cambio de signo en dos derivadas parciales consecutivas se identifica con un salto por encima de un mínimo, motivo por el que se realiza el retroceso. En iRprop⁺, el retroceso sólo se

⁴⁷⁹ J. M. Hannan y J. Mark Bishop. A Class of Fast Artificial NN Training Algorithms. Technical Report JMH-JMB 01/96, Department of Cybernetics, University of Reading, UK, 1996

⁴⁸⁰ Marcus Pfister y Raúl Rojas. Hybrid learning algorithms for feed-forward neural networks. En Bernd Reusch, editor, *Fuzzy Logik: Theorie und Praxis 4. Dortmunder Fuzzy-Tage Dortmund, 6.-8. Juni 1994*, pages 61–68, 1994. ISBN 978-3-642-79386-8. doi: 10.1007/978-3-642-79386-8_8

⁴⁸¹ Nicholas K. Treadgold y Tamas D. Gedeon. Simulated annealing and weight decay in adaptive learning: the SARPROP algorithm. *IEEE Transactions on Neural Networks*, 9(4):662–668, Jul 1998. ISSN 1045-9227. doi: 10.1109/72.701179

⁴⁸² Christian Igel y Michael Hüskens. Empirical evaluation of the improved Rprop learning algorithms. *Neurocomputing*, 50:105 – 123, 2003. ISSN 0925-2312. doi: 10.1016/S0925-2312(01)00700-7

efectúa si el cambio de signo ocasionó un aumento en el valor de la función de error, coste o pérdida. Si no, se considera positivo el paso ya dado y se continúa a partir de la posición actual. Sólo con almacenar el error previo cometido por la red, se puede combinar información individual sobre la superficie de error (el signo de la derivada parcial) con información global (el error observado) para decidir si se revierte o no un paso que ocasiona un cambio en el signo del gradiente.

La versión mejorada de Rprop, iRprop, suele funcionar mejor que Rprop y el uso de gradientes conjugados. Su rendimiento también es comparable a técnicas de segundo orden como el método BFGS.

- Las anteriores son sólo algunas muestras de las muchas variantes que se pueden diseñar de técnicas como Rprop. Se pueden encontrar versiones que funcionan sobre números complejos en vez de sobre números reales, como C-Rprop [*Complex Rprop*],⁴⁸³ de utilidad relativamente limitada en la práctica (por no ser demasiado severos). Otras modificaciones, como GRprop [*Globally-convergent Rprop*]⁴⁸⁴, se diseñan para que posean propiedades deseables desde el punto de vista matemático.

En el caso de GRprop, el algoritmo está diseñado de tal forma que se asegura que el ajuste de las tasas de aprendizaje locales garantice que se da un paso en la dirección del gradiente descendente en cada iteración del algoritmo.⁴⁸⁵ Esta característica le permite a GRprop converger más rápidamente que la versión original de Rprop o su versión mejorada, iRprop.

RMSprop

La magnitud del gradiente puede ser muy diferente para los distintos parámetros de una red. Este hecho dificulta la elección de una tasa de aprendizaje global y justifica el uso de Rprop, que sólo emplea el signo del gradiente y utiliza actualizaciones de magnitudes que se van ajustando dinámicamente. Los saltos se amplían (multiplicando por $\eta_0^+ = 1.2$) cuando se mantiene el signo del gradiente y se reducen (multiplicando por $\eta_0^- = 0.5$) cuando se observa un cambio en el signo del gradiente. Además, las magnitudes de los saltos se acotan (usando $\Delta_{min} = 10^{-6}$ y $\Delta_{max} = 50$).

Las características de Rprop nos permiten escapar fácilmente de zonas en las que el gradiente es reducido (p.ej. mesetas en la superficie de error). Sin embargo, el método sólo funciona realmente bien cuando empleamos aprendizaje por lotes, que no es lo más habitual en *deep learning*. Cuando se emplean minibatches en el gradiente descendente estocástico, la idea es utilizar tasas de aprendizaje pequeñas que permitan promediar los gradientes observados en minibatches sucesivos. Ahora bien, si observamos que un peso obtiene un gradiente igual a $+0.1$ para nueve minibatches consecutivos y -0.9 para el décimo que se encuentra, nos gustaría que

⁴⁸³ Bipin Kumar Tripathi y Prem Kumar Kalra. On efficient learning machine with root-power mean neuron in complex domain. *IEEE Transactions on Neural Networks*, 22(5):727–738, May 2011. ISSN 1045-9227. DOI: 10.1109/TNN.2011.2115251

⁴⁸⁴ Aristoklis D. Anastasiadis, George D. Magoulas, y Michael N. Vrahatis. New globally convergent training scheme based on the resilient propagation algorithm. *Neurocomputing*, 64: 253 – 270, 2005. ISSN 0925-2312. DOI: 10.1016/j.neucom.2004.11.016. Trends in Neurocomputing: 12th European Symposium on Artificial Neural Networks 2004

⁴⁸⁵ George D. Magoulas, Vassilis P. Plagianakos, y Michael N. Vrahatis. Globally convergent algorithms with local learning rates. *IEEE Transactions on Neural Networks*, 13(3):774–779, May 2002. ISSN 1045-9227. DOI: 10.1109/TNN.2002.1000148

el peso conservase su valor. ¿Qué es lo que haría Rprop en un caso así? Incrementar nueve veces su valor y reducirlo una sola vez. Dejando a un lado los efectos de la adaptación de la magnitud de las actualizaciones, que podemos asumir que se realizarían a una escala de tiempo más lenta, es probable que el valor del peso crezca demasiado.

El problema descrito en el párrafo anterior condujo al diseño de RMSprop [*Root-Mean Square backpropagation*], un método no publicado propuesto por Tijmen Tieleman y descrito por Geoffrey Hinton en su curso de Coursera.⁴⁸⁶ La idea de RMSprop es combinar la robustez de Rprop, la eficiencia del aprendizaje estocástico con minilotes y el promediado de gradientes sobre minilotes.

¿Cómo se consigue algo así? Si nos fijamos en Rprop, que sólo usa el signo del gradiente, podemos ver este signo como dividir el gradiente por su magnitud:

$$\text{sign}(g_i(t)) = \frac{g_i(t)}{|g_i(t)|}$$

El problema es que, cuando utilizamos minilotes, queremos dividir por diferentes valores para cada minilote. Lo que podemos hacer es calcular una medida que resulte similar para minilotes adyacentes. ¿Por ejemplo? Una media móvil:

$$E[g^2](t) = \gamma E[g^2](t-1) + (1-\gamma)g^2(t)$$

donde, una vez más, recurrimos a un hiperparámetro $\gamma = 0.9$ para controlar la media móvil.

Dividiendo el gradiente por la raíz cuadrada de esa media móvil obtenemos la fórmula de actualización de los pesos propuesta por RMSprop:

$$\Delta x(t) = -\frac{\eta}{\text{RMS}[g](t)} \odot g(t) = -\frac{\eta}{E[g^2](t) + \epsilon} \odot g(t)$$

El algoritmo RMSprop lo único que hace, en realidad, es cambiar la acumulación del gradiente de AdaGrad por una media móvil con suavizado exponencial.

Si nos fijamos bien, es el primer paso que dimos para derivar el método AdaDelta, sin la corrección posterior que realiza este último método para que las unidades coincidan (en términos físicos, que las ecuaciones sean dimensionalmente homogéneas). Igual que Matthew Zeiler, Tijmen Tieleman observó que se obtenían mejores resultados si dividíamos por la raíz cuadrada de la media móvil de los gradientes al cuadrado.

Ahora bien, a diferencia de lo que sucedía con AdaDelta, RMSprop sí que se puede combinar con el uso de momentos. Los momentos tradicionales no parecen contribuir demasiado a mejorar el rendimiento de RMSprop (y no se sabe muy bien por qué). Sin embargo, RMSprop sí que funciona bastante bien cuando se combina con momentos de Nesterov:

- RMSprop, sin momentos:

$$r(t) = \gamma r(t-1) + (1-\gamma) g(t) \odot g(t)$$

⁴⁸⁶ Geoffrey Hinton, Nitsh Srivastava, y Kevin Swersky. Neural networks for machine learning. *Coursera, MOOC video lectures*, 2012a

$$\Delta x(t) = -\frac{\eta}{\sqrt{r(t)} + \epsilon} \odot g(t)$$

- RMSprop con momentos de Nesterov:

1. Corrección de Nesterov:

$$\tilde{x}(t) = x(t) + \mu v(t-1)$$

2. Gradiente (sobre la posición corregida):

$$g(t) = \nabla f(\tilde{x}(t))$$

3. Media móvil:

$$r(t) = \gamma r(t-1) + (1-\gamma) g(t) \odot g(t)$$

4. Velocidad:

$$v(t) = \mu v(t-1) - \frac{\eta}{\sqrt{r(t)} + \epsilon} \odot g(t)$$

5. Actualización de los parámetros:

$$\Delta x(t) = x(t+1) - x(t) = v(t)$$

vSGD

En problemas de aprendizaje que trabajan con conjuntos de datos enormes, el aprendizaje por lotes no resulta efectivo. De ahí que se hayan diseñado múltiples estrategias que permitan ajustar automáticamente las tasas de aprendizaje cuando se utiliza el gradiente descendente estocástico [*SGD, stochastic gradient descent*].

Era el caso de RMSprop y también el de una técnica de la que daremos algunas pinceladas en esta sección: vSGD [*variance-based SGD*].⁴⁸⁷

vSGD es un método que ajusta automáticamente las tasas de aprendizaje, ya sea una global o tasas locales para cada parámetro, intentando optimizar una estimación del valor esperado para la función de error, coste o pérdida después de la siguiente actualización de los parámetros.

A partir de un escenario idealizado en el que las contribuciones al error de cada muestra son cuadráticas y separables, LeCun y sus colaboradores derivan una fórmula para las tasas de aprendizaje óptimas para el gradiente descendente estocástico:

$$\eta^*(t) = \frac{1}{h} \frac{(x(t) - x^*)^2}{(x(t) - x^*)^2 + \sigma^2}$$

La fórmula incluye dos términos. El primero captura la curvatura local de la función de error (asumiendo una matriz hessiana diagonal), mientras que el segundo captura la variabilidad que puede existir entre distintas muestras del conjunto de entrenamiento (la varianza del gradiente). Ambos pueden estimarse en la práctica de diferentes formas:

⁴⁸⁷ Tom Schaul, Sixin Zhang, y Yann LeCun. No more pesky learning rates. En Sanjoy Dasgupta y David McAllester, editores, *Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) de *Proceedings of Machine Learning Research*, pages 343–351, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/schaul13.html>

- vSGD-l [*vSGD local*] utiliza estimaciones de la varianza del gradiente local y de la diagonal de la matriz hessiana, con lo que la tasa de aprendizaje óptima es de la forma $\eta_i^* = g_i^2 / (h_i v_i)$, donde $g_i \approx E[\nabla_{x_i}]$, $v_i \approx E[\nabla_{x_i}^2]$ y h_i es el i-ésimo elemento de la diagonal de la matriz hessiana.
- vSGD-g [*vSGD global*] utiliza una estimación global del gradiente de la varianza y una cota superior para los términos de la diagonal de la matriz hessiana para obtener $\eta^* = \sum g_i^2 / (h^+ l)$, donde $h^+ = \max\{h_i\}$ y $l \approx E[||\nabla_x||^2]$.

Todos los parámetros involucrados se pueden calcular de forma eficiente, en tiempo y espacio lineal, usando medias móviles con un suavizado exponencial: el cuadrado de los gradientes esperados g , el valor esperado de los gradientes al cuadrado v , el valor esperado de la norma al cuadrado del gradiente l y el valor esperado de la diagonal de la matriz hessiana h . Este último valor esperado se obtiene mediante un algoritmo llamado *bb-prop*.⁴⁸⁸ El algoritmo *bbprop* utiliza una aproximación de Gauss-Newton para obtener una estimación de cada elemento de la diagonal de la matriz hessiana, para lo que se multiplica la curvatura de la función por el cuadrado de la derivada de la función de coste con respecto a cada parámetro. El uso de *bbprop* implica un recorrido adicional de la red con *backpropagation* y está limitado a funciones con curvatura; es decir, no se puede utilizar para funciones de coste de tipo L_1 . Una variante posterior, vSGD-fd [*vSGD with finite-difference-estimated adaptive learning rates*],⁴⁸⁹ obtiene una estimación de cada elemento de la diagonal de la matriz hessiana a partir de dos gradientes y no tiene la limitación de la estimación anterior. Para estimar la diagonal de la matriz hessiana usando diferencias finitas se utiliza la siguiente expresión:

$$h_i^{fd} = \left| \frac{\nabla_{x_i} - \nabla_{x_i+g_i}}{g_i} \right|$$

Sin necesidad de ajustar manualmente ningún hiperparámetro, vSGD consigue resultados comparables a los mejores resultados posibles que se pueden obtener utilizando otras estrategias de adaptación de las tasas de aprendizaje. Cuando se usa con aprendizaje *online*, las tasas de aprendizaje aumentan automáticamente cuando se producen cambios en la distribución de los datos recibidos y se reducen dinámicamente cuando la distribución de los datos se mantiene estacionaria, lo que hace al método vSGD especialmente robusto frente a cambios dinámicos.

AdaSecant

Tras haber comentado las técnicas de adaptación de las tasas de aprendizaje propuestas por los grupos de Geoffrey Hinton en la Universidad

⁴⁸⁸ Yann LeCun, Leon Bottou, Genevieve B. Orr, y Klaus Robert Müller. Efficient BackProp. En Genevieve B. Orr y Klaus-Robert Müller, editores, *Neural Networks: Tricks of the Trade*, pages 9–50. Springer, 1998b. ISBN 3540653112. DOI: 10.1007/3-540-49430-8_2

⁴⁸⁹ Tom Schaul y Yann LeCun. Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients. *ICLR’2013*, arXiv:1301.3764, 2013. URL <http://arxiv.org/abs/1301.3764>

de Toronto (RMSprop) y de Yann LeCun en la Universidad de Nueva York (vSGD), no podíamos pasar por alto la propuesta al respecto que realiza el grupo de Yoshua Bengio en la Universidad de Montréal: AdaSecant.^{490,491}

Igual que en vSGD, AdaSecant utiliza información relativa a la curvatura de la función de coste (su segunda derivada, dada por la matriz hessiana). Ahora bien, en vez de estimar los valores de la diagonal de la matriz hessiana usando el método de Gauss-Newton o diferencias finitas, que resultan en estimaciones de h_i demasiado sensibles al ruido en la estimación estocástica de los gradientes g_i , AdaSecant estima la curvatura local de la función de coste en la dirección del gradiente. Además, propone el uso de una técnica de reducción de la varianza en la estimación del gradiente para acelerar la convergencia del algoritmo.

Para determinar el tamaño óptimo del paso, AdaSecant utiliza el método de la secante, de donde proviene su nombre. Igual que en otros métodos, las estimaciones de los diferentes parámetros se realizan de forma eficiente mediante el uso de medias móviles. El resultado final es el siguiente:

- Se realiza una estimación robusta del gradiente descendente estocástico, utilizando la siguiente transformación para reducir la varianza del gradiente estocástico g_i :

$$\tilde{g}_i = \frac{g_i + \gamma_i E[g_i]}{1 + \gamma_i}$$

donde γ_i es un número real positivo que hace que se mantenga el valor esperado del gradiente, $E[\tilde{g}_i] = E[g_i]$, pero se reduzca su varianza en un factor $(1 + \gamma_i)^2$.

Como no conocemos el valor real de $E[g_i]$, lo aproximamos a partir de los valores pasados de g_i con una media móvil y adaptamos dinámicamente el valor del parámetro γ_i usando la expresión

$$\gamma_i = \frac{E[(g_i - g'_i)(g_i - E[g_i])]}{E[(g_i - E[g_i])(g'_i - E[g_i])] + \lambda}$$

donde g'_i hace referencia al gradiente estocástico del siguiente minilote y λ es un coeficiente de regularización.

- En la aproximación de la secante del método direccional de Newton, la tasa de aprendizaje para el parámetro x_i se estima utilizando diferencias finitas:

$$\eta_i = \frac{\Delta x_i}{\nabla_{x_i} f(x + \Delta x) - \nabla_{x_i} f(x)}$$

En el caso estocástico, aproximamos su valor esperado a partir de distintos minilotes:

$$E[\eta_i] = E \left[\frac{\Delta x_i}{\nabla_{x_i} f(x + \Delta x) - \nabla_{x_i} f(x)} \right]$$

⁴⁹⁰ Çağlar Gülgahre, Marcin Moczulski, y Yoshua Bengio. ADASECANT: robust adaptive secant method for stochastic gradient. *arXiv e-prints*, arXiv:1412.7419, 2014. URL <http://arxiv.org/abs/1412.7419>

⁴⁹¹ Çağlar Gülgahre, José Sotelo, Marcin Moczulski, y Yoshua Bengio. A robust adaptive stochastic gradient method for deep learning. En *IJCNN'2017 International Joint Conference on Neural Networks*, pages 125–132, May 2017. doi: 10.1109/IJCNN.2017.7965845

Como la aproximación anterior resulta numéricamente inestable, se recurre a una aproximación en serie de Taylor de segundo orden en torno a $(\sqrt{E[\alpha_i^2]}, \sqrt{E[\Delta x_i^2]})$, donde α_i el denominador de la expresión anterior: $\alpha_i = \nabla_{x_i} f(x + \Delta x) - \nabla_{x_i} f(x)$.

Asumiendo que $\sqrt{E[\alpha_i^2]} \approx E[\alpha_i]$ y que $\sqrt{E[\Delta x_i^2]} \approx E[\Delta x_i]$, se obtiene una aproximación no negativa de la tasa de aprendizaje:

$$E[\eta_i] \approx \frac{\sqrt{E[\Delta x_i^2]}}{\sqrt{E[\alpha_i^2]}} - \frac{\text{Cov}(\alpha_i, \Delta x_i)}{E[\alpha_i^2]}$$

La covarianza, que aparece en el segundo término de la aproximación en serie de Taylor, se puede aproximar de forma más simple como $E[\alpha_i \Delta x_i]$, con lo que obtenemos la expresión final de la tasa de aprendizaje utilizada por AdaSecant:

$$\eta_i = \frac{\sqrt{E[\Delta x_i^2]}}{\sqrt{E[\alpha_i^2]}} - \frac{E[\alpha_i \Delta x_i]}{E[\alpha_i^2]}$$

- Una vez calculado el gradiente estocástico, ajustado para reducir su varianza, y la tasa de aprendizaje idónea, actualizamos los valores de los parámetros como en cualquier otro método basado en el gradiente descendente:

$$\Delta x_i = -\eta_i \cdot \tilde{g}_i$$

Los dos puntos clave del algoritmo radican en: (1) determinar la curvatura de la función en la dirección del gradiente usando el método de la secante, para lo que hay que evaluar el gradiente en dos puntos, y (2) reducir la varianza de la estimación estocástica del gradiente. Todos los valores necesarios para realizar esos cálculos se aproximan mediante el uso de medias móviles, igual que en otras variantes del gradiente descendente que ya hemos visto.

Como AdaDelta o vSGD, AdaSecant no necesita el ajuste manual de ningún hiperparámetro, lo que siempre resulta positivo cuando hablamos de técnicas de aprendizaje automático. El método de optimización AdaSecant se puede encontrar en herramientas de *deep learning* como Theano o Pylearn2, que está construida sobre la primera.

Adam

Para cerrar nuestro repaso de técnicas de ajuste adaptativo de las tasas de aprendizaje, describiremos otro algoritmo frecuente en *deep learning* que utiliza momentos adaptativos: Adam [*Adaptive moment estimation*].⁴⁹²

⁴⁹² Diederik P. Kingma y Jimmy Ba. Adam: A Method for Stochastic Optimization. *ICLR'2015*, arXiv:1412.6980, 2014. URL <http://arxiv.org/abs/1412.6980>

Ya vimos que RMSprop se podía combinar con momentos de Nesterov. Sin embargo, tal como se propuso originalmente, AdaDelta se tenía que utilizar sin momentos. Adam es una variante de la combinación de RMSprop con momentos que se puede considerar una extensión de AdaDelta en la que se incorpora el uso de momentos.

En Adam se mantienen dos medias móviles de los gradientes:

$$\begin{aligned} m(t) &= \beta_1 m(t-1) + (1 - \beta_1) g(t) \\ v(t) &= \beta_2 v(t-1) + (1 - \beta_2) g^2(t) \end{aligned}$$

donde $m(t)$ es una estimación del primer momento del gradiente (su media) y $v(t)$ es una estimación del segundo momento del gradiente (su varianza). Los parámetros que controlan las medias móviles, β_1 y β_2 , tienen valores cercanos a 1, p.ej. $\beta_1 = 0.9$ y $\beta_2 = 0.999$.

Como las estimaciones se inicializan a cero y las tasas de suavizado exponencial se mantienen pequeñas, las estimaciones $m(t)$ y $v(t)$ están sesgadas hacia 0, motivo por el que Adam corrige el sesgo de las estimaciones anteriores utilizando

$$\begin{aligned} \hat{m}(t) &= \frac{m(t)}{1 - \beta_1^t} \\ \hat{v}(t) &= \frac{v(t)}{1 - \beta_2^t} \end{aligned}$$

A partir de los valores anteriores se deriva la fórmula de actualización de los valores de los parámetros utilizada en Adam:

$$\Delta x = -\frac{\eta}{\sqrt{\hat{v}(t)} + \epsilon} \hat{m}(t)$$

donde η es una tasa de aprendizaje inicial (un hiperparámetro del algoritmo, p.ej. $\eta = 0.002$), ϵ se utiliza para evitar divisiones por cero ($\epsilon = 10^{-8}$), $\hat{m}(t)$ ejerce de gradiente y $\hat{v}(t)$, el segundo momento del gradiente, ajusta dinámicamente la tasa de aprendizaje efectiva.

En Adam, $m(t)$ se calcula como una media móvil de los gradientes de los últimos minilotes. Al utilizarse en el lugar del gradiente en la expresión del gradiente descendente, conseguimos incorporar momentos al gradiente descendente de forma directa.

Si eliminásemos la media móvil $m(t)$ de Adam y utilizásemos directamente el gradiente de cada minilote, nos quedaríamos básicamente con AdaDelta, de ahí que podamos ver Adam como una versión de AdaDelta (que ajusta la tasa de aprendizaje utilizando información de segundo orden) con momentos (suavizando los gradientes de minilotes consecutivos mediante una media móvil).

Si, por el contrario, suprimiésemos la media móvil $v(t)$, tendríamos, esencialmente, la versión tradicional del gradiente descendente estocástico con momentos.

Adam suele considerarse bastante robusto con respecto a los valores iniciales de sus hiperparámetros (η , β_1 y β_2), aunque en ocasiones tendremos que ajustar manualmente la tasa de aprendizaje de referencia, η .

En el mismo artículo en el que se propuso Adam, también se incluyó una variante denominada AdaMax. Y dado que Adam utiliza momentos tradicionales pero ya hemos visto que muchas veces se utilizan momentos de Nesterov, tampoco debería sorprendernos demasiado que también se haya propuesto una extensión de Adam con momentos de Nesterov, denominada NAdam. Veamos en qué consisten ambas variantes:

- *AdaMax:*

En Adam, la media móvil $v(t)$ se calcula utilizando la norma L_2 de los gradientes observados en los minibatches anteriores. Podemos generalizar esa media móvil para una norma cualquier L_p :

$$v(t) = \beta_2^p v(t-1) + (1 - \beta_2^p) |g(t)|_p$$

Las normas con valores grandes de p suelen ser numéricamente inestables, de ahí que casi siempre utilicemos las normas L_1 o L_2 . Sin embargo, la norma L_∞ , que corresponde a la distancia de Chebyshev, más conocida como distancia del tablero de ajedrez, también es numéricamente estable. Si utilizamos esta norma, podemos calcular la media móvil

$$u(t) = \beta_2^\infty u(t-1) + (1 - \beta_2^\infty) |g(t)|_p = \max\{\beta_2 \cdot u(t-1), |g(t)|\}$$

Utilizando esta media móvil en la expresión de Adam, obtenemos la regla de actualización de AdaMax:

$$\Delta x = -\frac{\eta}{u(t)} \hat{m}(t)$$

donde ya no hace falta que corrijamos la estimación de $u(t)$, al no estar sesgada hacia 0.

- *NAdam [Nesterov-accelerated adaptive moment estimation].*⁴⁹³

Igual que Adam es RMSprop con momentos tradicionales, NAdam es RMSprop con momentos de Nesterov. Aplicando la estrategia habitual de los momentos de Nesterov (en los que se evalúa el gradiente tras realizar la corrección de Nesterov), se puede llegar a la siguiente expresión de actualización:

$$\Delta x = -\frac{\eta}{\sqrt{\hat{v}(t)} + \epsilon} \left(\beta_1 \hat{m}(t) + \frac{(1 - \beta_1) g(t)}{1 - \beta_1^t} \right)$$

donde se reemplaza el término $\hat{m}(t)$ de Adam por la expresión, algo más compleja, que aparece entre paréntesis.

⁴⁹³ Timothy Dozat. Incorporating Nesterov Momentum into Adam. *ICLR'2016 Workshop Track*, 2015. URL http://cs229.stanford.edu/proj2015/054_report.pdf

Hemos visto multitud de algoritmos diferentes que nos permiten resolver problemas de optimización utilizando el gradiente descendente, muchas de ellas diseñadas específicamente para su uso con minilotes en *deep learning* (gradiente descendente estocástico). A menudo, las diferentes técnicas heurísticas propuestas se pueden hibridar unas con otras para conseguir variantes proximales con regularización,⁴⁹⁴ como AdaGrad-RDA [*regularized dual averaging*]⁴⁹⁵ o distintas variantes de la familia FTRL [*follow-the-regularized-leader*] para el aprendizaje *online*.⁴⁹⁶ En otras ocasiones, se proponen extensiones y nuevas heurísticas para adaptar las tasas de aprendizaje dinámicamente, como en el gradiente descendente equilibrado.⁴⁹⁷ El número de posibilidades es prácticamente infinito, como lo atestigua el hecho de que se lleven proponiendo variantes del gradiente descendente desde que Cauchy lo utilizó en 1847 y del gradiente descendente estocástico desde los trabajos sobre aproximación estocástica de Robbins y Monro en 1951.

Ante tal gama de técnicas disponibles, siempre nos quedarán dudas a la hora de decidir cuál puede ser mejor para el problema concreto al que nos enfrentemos. A veces, las limitaciones de las herramientas que utilicemos nos obligarán a utilizar técnicas específicas, que no siempre serán las más adecuadas. En el mejor de los casos, puede que tengamos la oportunidad de realizar una comparación empírica de las técnicas que tengamos a nuestra disposición, tal como ya han realizado algunos investigadores.⁴⁹⁸ Normalmente, no habrá vencedores claros y la elección de qué algoritmo concreto utilizar dependerá, en gran medida, de la familiaridad que ya tengamos con él y de la facilidad con la que podamos ajustar sus hiperparámetros.

Técnicas de optimización de segundo orden

Hemos visto que muchas de las técnicas que se utilizan para ajustar automáticamente las tasas de aprendizaje en el gradiente descendente intentan estimar de alguna forma propiedades de la segunda derivada de la función de coste o pérdida. Los métodos de optimización que, de forma explícita, utilizan la segunda derivada se denominan métodos de optimización de segundo orden, para diferenciarlos de los métodos de optimización de primer orden, que sólo emplean el gradiente.⁴⁹⁹

Para entender en qué se basan y cómo diseñar técnicas de optimización de segundo orden, hemos de introducir algo de notación. Hasta ahora hemos venido utilizando funciones escalares de varias variables, de la forma $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Para estas funciones multivariable, podemos definir el gradiente como la generalización del concepto de derivada en funciones escalares de una única variable. El gradiente no es más que el vector de las derivadas parciales de la función con respecto a cada una de sus

⁴⁹⁴ Yoram Singer y John C. Duchi. Efficient Learning using Forward-Backward Splitting. En Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, y A. Culotta, editores, *NIPS'2009 Advances in Neural Information Processing Systems 22*, pages 495–503. Curran Associates, Inc., 2009. URL <https://goo.gl/cnhvT4>

⁴⁹⁵ John Duchi, Elad Hazan, y Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, July 2011. ISSN 1532-4435. URL <http://jmlr.org/papers/v12/duchi11a.html>

⁴⁹⁶ H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, y Jeremy Kubica. Ad Click Prediction: A View from the Trenches. En *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1222–1230, 2013. ISBN 978-1-4503-2174-7. DOI: 10.1145/2487575.2488200

⁴⁹⁷ Yann Dauphin, Harm de Vries, y Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization. En C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, y R. Garnett, editores, *NIPS'2015 Advances in Neural Information Processing Systems 28*, pages 1504–1512. Curran Associates, Inc., 2015. URL <https://goo.gl/FJo8Sv>

⁴⁹⁸ Tom Schaul, Ioannis Antonoglou, y David Silver. Unit tests for stochastic optimization. *ICLR'2014*, arXiv:1312.6055, 2014. URL <http://arxiv.org/abs/1312.6055>

⁴⁹⁹ Jorge Nocedal y Stephen Wright. *Numerical Optimization*. Springer, 2nd edition, 2006. ISBN 0387303030

variables:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right]$$

Cuando usamos las segundas derivadas de una función multivariable, trabajamos con funciones vectoriales de varias variables, de la forma $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Esto es, una función vectorial de \mathbb{R}^n en \mathbb{R}^m podemos verla como un conjunto de m funciones escalares f_i tales que $f(x) = (f_1(x), f_2(x), \dots, f_m(x))$.

Dada una función vectorial multivariable f , podemos definir su matriz jacobiana $J(f)$ a partir del conjunto de derivadas parciales de cada una de las funciones escalares de la que se compone con respecto a sus variables:

$$J(f) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \dots & \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Esta matriz jacobiana recibe su nombre del matemático alemán Carl Gustav Jacob Jacobi.

Para hacer referencia a un elemento concreto de la matriz jacobiana, utilizaremos la notación J_{ij} :

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

El determinante de la matriz jacobiana evaluada en un punto x , denominado jacobiano, nos informa acerca del comportamiento de la función f en torno a x : su valor absoluto nos da el factor con el que la función f expande o contrae su volumen cerca de x .

Cuando $m = 1$, la función f es escalar y la matriz jacobiana se reduce a un vector, el gradiente de la función. Es decir, el jacobiano generaliza el gradiente igual que el gradiente generaliza la noción de derivada. Si la función f es diferenciable en el punto p , la matriz jacobiana se puede emplear para definir una función $\mathbb{R}^n \rightarrow \mathbb{R}^m$ que es la mejor aproximación lineal de f cerca del punto p :

$$f(x) \approx f(p) + J_f(p)(x - p)$$

exactamente igual que cuando utilizamos la derivada para aproximar el valor de una función en torno a un punto.

Una vez definida la matriz jacobiana de una función vectorial, pasamos a definir la matriz hessiana de una función escalar. La matriz hessiana $H(f)$ se define a partir del conjunto de las segundas derivadas de la función escalar multivariable, $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

El uso de la matriz hessiana fue propuesto en el siglo XIX por el matemático alemán Ludwig Otto Hesse. Aunque Hesse utilizó originalmente la denominación de “determinantes funcionales”, posteriormente se denominó matriz hessiana en su honor. Como la segunda derivada de una función escalar univariable, la matriz hessiana describe la curvatura local de una función multivariable.

Como hicimos con la matriz jacobiana, para hacer referencia a un elemento concreto de la matriz hessiana emplearemos la notación H_{ij} :

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

La matriz hessiana es simétrica para funciones cuyas segundas derivadas parciales sean continuas: $H_{ij} = H_{ji}$.

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$$

La matriz hessiana está relacionada con la matriz jacobiana, ya que podemos ver la matriz hessiana como el jacobiano del gradiente:

$$H(f) = J(\nabla f)^\top$$

Es importante no confundir la matriz hessiana con el operador de Laplace, también llamado laplaciano, que se define como la divergencia del gradiente (la suma de las segundas derivadas parciales con respecto a cada variable independiente x_i):

$$\nabla^2 f = \nabla \cdot \nabla f = \sum_{i=1}^n \frac{\partial^2 f}{\partial x_i^2}$$

Mientras que el hessiano de una función escalar es la matriz cuadrada de las segundas derivadas de la función, el laplaciano es un valor escalar. Este valor se calcula usando un operador vectorial denominado divergencia, que tal vez recuerde de sus clases de Física de secundaria, donde seguramente escucharía hablar por primera vez del gradiente, de la divergencia y del rotacional en la descripción de campos vectoriales.

El hessiano de una función vectorial $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ se puede definir fácilmente a partir de los componentes de la función vectorial. Dada la función $f(x) = (f_1(x), f_2(x), \dots, f_m(x))$, su colección de segundas derivadas será de la forma $H(f) = (H(f_1), H(f_2), \dots, H(f_m))$, por lo que el hessiano será un tensor de tercer orden de tamaño $m \times n \times n$ en lugar de una matriz de tamaño $n \times n$. En términos informáticos, en vez de trabajar con un *array* bidimensional (la matriz hessiana), trabajaremos con un *array* tridimensional (el hessiano de una función vectorial).

- La divergencia, $\operatorname{div} f = \nabla \cdot f$, produce un escalar que, desde el punto de vista físico, mide la diferencia entre el flujo saliente y el flujo entrante de un campo vectorial sobre la superficie que rodea a un volumen. Será positiva si, en términos netos, el volumen incluye fuentes del campo vectorial y negativa si contiene más sumideros que fuentes. La divergencia en un punto es el flujo del campo vectorial por unidad de volumen cuando el volumen en torno al punto tiende a cero.

Imaginemos un volumen de aire, cuya velocidad en cada punto define un campo vectorial. Cuando el aire de una zona se calienta, se expande en todas direcciones, sus vectores de velocidad apuntan hacia afuera y la divergencia del campo es positiva. Cuando el aire se enfriá, se contrae y la divergencia de la velocidad es negativa.

- El rotacional, $\operatorname{rot} f = \nabla \times f$, es otro operador vectorial, definido sobre campos vectoriales tridimensionales y difícilmente generalizable a más dimensiones. Mide la tendencia de un campo vectorial a rotar alrededor de un punto. James Clerk Maxwell lo denominó *curl* [rizo] en 1871.

Si imaginamos un campo vectorial que describe la velocidad de un fluido en el que dejamos una pequeña bola fija en un punto, el fluido hará rotar la bola y su eje de rotación, de acuerdo a la regla de la mano derecha, apuntará en la dirección del rotacional del campo en el punto correspondiente al centro de la bola. La magnitud del rotacional será proporcional a la velocidad angular de la rotación.

Las ecuaciones de Maxwell, que describen el electromagnetismo, se pueden enunciar de forma muy elegante en términos de la divergencia y del rotacional de los campos eléctrico E y magnético B :

$$\begin{aligned}\nabla \cdot E &= \frac{\rho}{\epsilon_0} & \nabla \times E &= -\frac{\partial B}{\partial t} \\ \nabla \cdot B &= 0 & \nabla \times B &= \mu_0 \left(J + \epsilon_0 \frac{\partial E}{\partial t} \right)\end{aligned}$$

donde se puede ver que la divergencia del campo eléctrico es proporcional a la densidad de carga y siempre cero para el campo magnético. La Ley de Faraday implica que, si el rotacional de un campo eléctrico no es cero, entonces existe un campo magnético variable. Por último, cuando el rotacional del campo magnético es distinto de cero, el campo magnético puede estar originado por una corriente eléctrica (Ley de Ampère) o por un campo eléctrico variable (extensión de Maxwell).

A diferencia de lo que sucede con los campos electromagnéticos, el campo gravitatorio es conservativo: su rotacional es siempre cero.

$$\begin{aligned}\nabla \cdot g &= -4\pi G\rho \\ \nabla \times g &= 0\end{aligned}$$

Las ecuaciones de Maxwell incluyen las leyes de Gauss para campos eléctricos (flujo proporcional a la carga en el interior de un volumen) y magnéticos (flujo cero, lo que implica la no existencia de monopolos magnéticos), la ley de Faraday de la inducción electromagnética (voltaje inducido en un bucle cerrado proporcional al cambio del flujo magnético) y la ley de Ampère (si circula una corriente por un conductor, se induce un campo magnético rotacional a su alrededor).

La ley de Gauss para el campo gravitacional es equivalente a la ley de la gravitación universal de Newton, igual que la ley de Gauss para campos eléctricos es equivalente a la ley de Coulomb.

En un campo conservativo, podemos describir el campo vectorial por un campo escalar: el potencial gravitatorio P en el caso del campo gravitatorio, del que el campo vectorial es su gradiente, $\mathbf{g} = -\nabla P$. El campo electrostático también es conservativo, por lo que puede describirse por un escalar, $E = -\nabla V$, el potencial eléctrico V medido en voltios.

El laplaciano aparece en las ecuaciones diferenciales con las que se describen numerosos fenómenos físicos, como las ecuaciones de Poisson para los potenciales eléctricos ($-\nabla^2 V = \rho/\epsilon_0$) y gravitatorios ($\nabla^2 P = 4\pi G\rho$). También aparece al describir la difusión del calor o la propagación de ondas. En el caso de las ondas electromagnéticas, sus ecuaciones se pueden derivar directamente de las leyes de Maxwell utilizando la siguiente identidad: $\nabla \times (\nabla \times \mathbf{X}) = \nabla(\nabla \cdot \mathbf{X}) - \nabla^2 \mathbf{X}$.

Con la ayuda de la notación adecuada, podemos representar, en apenas unas líneas, las leyes que gobiernan las fuerzas fundamentales de la naturaleza (gravedad y electromagnetismo, en particular).

Una vez familiarizados con la terminología específica del cálculo multivariable, volvamos al problema que nos ocupa, la resolución de problemas de optimización más allá del gradiente descendente, para lo que recurriremos a la matriz hessiana.

Igual que la segunda derivada, la matriz hessiana nos da información acerca de la curvatura local de una función. En el caso univariable, la segunda derivada nos indica si la función $f(x)$ no tiene curvatura (es una función lineal, $f''(x) = 0$), tiene curvatura positiva (es una función convexa, curvada hacia arriba, $f''(x) > 0$) o tiene curvatura negativa (es una función cóncava, curvada hacia abajo, $f''(x) < 0$). Cuando nos encontramos ante un punto crítico, para el cual la derivada de la función es cero ($f'(x) = 0$), el signo de la segunda derivada nos permite determinar el tipo de punto crítico ante el que nos encontramos: si es positiva, $f''(x) > 0$, estamos ante un mínimo local; si es negativa, $f''(x) < 0$, se trata de un máximo local; si es nula, $f''(x) = 0$, puede ser un punto de silla o, simplemente, una zona plana de la función.

De manera análoga, podemos recurrir a la matriz hessiana de una función multivariable para determinar el tipo de un punto crítico. En este caso, los puntos críticos son los puntos para los que el gradiente es nulo ($\nabla_x f(x) = 0$). Las propiedades de la matriz hessiana evaluada en el punto crítico nos ayudarán a determinar el tipo de los puntos críticos de la función. Como tenemos una matriz en vez de un valor escalar, recurriremos al signo de sus autovalores [*eigenvalues*]:

- Si la matriz hessiana es definida positiva (todos sus autovalores o *eigenvalues* son positivos), estamos en un mínimo local de la función.
- Si la matriz hessiana es definida negativa (todos sus autovalores o *eigenvalues* son negativos), nos encontramos antes un máximo local

En álgebra lineal, los vectores propios, autovectores o *eigenvectores* de una matriz son los vectores que, cuando les aplicamos la matriz como un operador lineal, dan lugar a un múltiplo escalar de sí mismos, con lo que no cambian de dirección: $Av = \lambda v$. El escalar λ recibe el nombre de valor propio o autovalor.

de la función.

- Cuando existen autovalores tanto negativos como positivos, se trata de un punto de silla, que corresponde a un máximo local en algunas direcciones y a un mínimo local en otras direcciones.
- Si existe algún autovalor nulo, no sabemos de qué tipo de punto crítico se trata.

En el caso univariable, la segunda derivada $f''(x)$ nos indica cómo cambia la primera derivada $f'(x)$ conforme variamos la entrada x , lo que nos puede ayudar a determinar el tamaño del salto que deberíamos dar para acercarnos lo máximo posible a un óptimo de la función $f(x)$. En el caso multivariante, el gradiente ∇f desempeña el papel de la primera derivada y la matriz hessiana $H(f)$, el de la segunda derivada. Los métodos de optimización de segundo orden, por tanto, utilizarán información obtenida a partir de la matriz hessiana de la función.

El cálculo de la matriz hessiana puede incorporarse en el algoritmo de propagación de errores, *backpropagation*.⁵⁰⁰ Sin embargo, desde el punto de vista numérico, el cálculo de la matriz hessiana en una red multicapa es lo que los matemáticos denominan un problema mal condicionado: un problema cuya solución es extremadamente sensible a pequeños cambios en los valores de sus coeficientes y, por tanto, propenso a errores numéricos debidos a la representación discreta de los números reales en un ordenador digital.⁵⁰¹ Es más, cuando tenemos un número elevado de variables, como es habitual en *deep learning*, el tamaño de la matriz hessiana, cuadrático con respecto al número de variables, hace que ni siquiera resulte viable representarla en memoria. Tendremos que idear métodos de optimización que sean capaces de aprovechar la información que nos podría proporcionar la matriz hessiana, pero sin llegar a calcularla realmente.

Cuando utilizábamos técnicas de optimización de primer orden, basados en el gradiente, nos encontrábamos con el problema de determinar adecuadamente los parámetros que gobernaban el entrenamiento de la red (las tasas de aprendizaje, en particular). Las técnicas de segundo orden nos ayudarán a ajustar automáticamente esos parámetros. No obstante, dada la sensibilidad numérica del cálculo de las segundas derivadas, nos veremos normalmente obligados a utilizar aprendizaje por lotes, en lugar del aprendizaje estocástico (*online* o con minilotes). Al utilizar técnicas de segundo orden, estimaremos gradiente y segundas derivadas a partir del conjunto de entrenamiento completo en vez de utilizar las estimaciones estocásticas del gradiente empleadas por el gradiente descendente estocástico.

A grandes rasgos, podríamos clasificar las técnicas de optimización de segundo orden en técnicas matriciales o vectoriales:

⁵⁰⁰ Christopher M. Bishop. Exact Calculation of the Hessian Matrix for the Multilayer Perceptron. *Neural Computation*, 4(4):494–501, 1992. DOI: 10.1162/neco.1992.4.4.494

⁵⁰¹ Sirpa Saarinen, Randall Bramley, y George Cybenko. Ill-conditioning in neural network training problems. *SIAM Journal on Scientific Computing*, 14(3):693–714, 1993. DOI: 10.1137/0914044

- Los algoritmos matriciales requieren almacenar la matriz hessiana (y su inversa). Es el caso del método de Newton o del algoritmo BFGS [*Broyden-Fletcher-Goldfarb-Shanno*]. Los algoritmos matriciales son típicamente dos órdenes de magnitud más rápidos que el gradiente descendente usando con *backpropagation*, si bien su complejidad computacional es, al menos, de orden $O(n^2)$, lo que los hace inviables para entrenar redes neuronales de cierto tamaño.
- Los algoritmos vectoriales sólo almacenan algunos vectores, como es el caso de L-BFGS, una variante de BFGS con uso limitado de memoria [*limited-memory BFGS*]; el método de la secante en un paso, OSS [*one-step secant*]; o los diferentes algoritmos basados en el uso de gradientes conjugados, CG [*conjugate gradient*]. Usualmente, son un orden de magnitud más rápidos que el gradiente descendente con *backpropagation* y sólo requieren la realización de cálculos iterativos capaces de explotar, de forma implícita, la estructura de la matriz hessiana.

El método de Newton

El método de optimización de Newton es un algoritmo iterativo basado en realizar una expansión en serie de Taylor para aproximar el valor de la función $f(x)$ en el entorno de un punto x_0 . Si realizamos una expansión de segundo orden, obtenemos la siguiente aproximación del valor de la función $f(x)$:

$$f(x) \approx f(x_0) + (x - x_0)\nabla_x f(x_0) + \frac{1}{2}(x - x_0)H_f(x_0)(x - x_0)$$

Queremos encontrar un punto crítico de la función x^* , en el cual se anule el gradiente $\nabla_x f(x^*)$. Si definimos $\Delta x = x - x_0$, derivamos con respecto a Δx e igualamos a cero para obtener el punto crítico de la función:

$$x^* = x_0 - H_f^{-1}(x_0)\nabla_x f(x_0)$$

Supongamos que la matriz hessiana H_f es definida positiva, por lo que nos encontramos en el entorno de un mínimo local de la función. Para una función f localmente cuadrática, al escalar el gradiente por la inversa de la matriz hessiana, el método de Newton salta directamente a ese mínimo local. Si la función objetivo es convexa pero no cuadrática (existen otros términos de orden superior en su aproximación en serie de Taylor), la actualización anterior se puede repetir de forma iterativa:

$$x_{n+1} = x_n - H_f^{-1}(x_n)\nabla_x f(x_n)$$

Expresando la ecuación anterior de la forma en la que definíamos el gradiente descendente, podemos ver cómo la inversa de la matriz hessiana

es la que determina, de forma automática, la tasa óptima de aprendizaje del algoritmo iterativo de optimización:

$$\Delta x = x_{n+1} - x_n = -H_f^{-1}(x_n)\nabla_x f(x_n)$$

donde la tasa de aprendizaje se elige dinámicamente dependiendo del punto de la función en el que nos encontramos, usando $H_f^{-1}(x_n)$.

Normalmente, en la implementación del algoritmo iterativo de optimización se suele incluir una constante $\gamma \in (0, 1)$, menor que 1, por lo que la regla de actualización será de la forma:

$$\Delta x = x_{n+1} - x_n = -\gamma H_f^{-1}(x_n)\nabla_x f(x_n)$$

donde el parámetro γ se elige para asegurar el cumplimiento de las condiciones de Wolfe, que garantizan la convergencia del algoritmo iterativo.

Geométricamente, el efecto de multiplicar por la inversa de la matriz hessiana es equivalente a eliminar las correlaciones existentes entre las diferentes variables, reduciendo la matriz de covarianza a una matriz identidad, lo que permite que el gradiente apunte siempre en la dirección correcta hacia el mínimo (en funciones cuadráticas o funciones que puedan aproximarse localmente por una función cuadrática, evidentemente). El efecto conseguido es, en cierto modo, similar al que se consigue utilizando la distancia de Mahalanobis en lugar de la distancia euclídea. La distancia de Mahalanobis es una generalización multidimensional de la medida de distancia basada en el número de desviaciones estándar con respecto a la media (variables tipificadas o *z-scores*). Equivale a la distancia euclídea una vez transformados los datos para que su matriz de covarianza sea la matriz identidad.

El método de Newton encuentra, de forma iterativa, un punto crítico de la función de coste o pérdida que pretendemos minimizar, un punto en el que su gradiente es nulo. Ahora bien, nada nos garantiza que la solución devuelta sea realmente un mínimo local de la función, que es lo que realmente nos interesa. Dado que conforme aumenta la dimensionalidad del problema, es mucho más probable que nos encontremos con puntos de silla que con mínimos locales, los puntos de silla suponen un serio problema para el correcto funcionamiento del método de Newton. Sin realizar las modificaciones oportunas, seguramente estemos encontrando un punto de silla y no un mínimo de la función.

Yann Dauphin y otros miembros del equipo de Yoshua Bengio en la Universidad de Montréal propusieron una variante del método de Newton que evita los puntos de silla y mejora los resultados que se obtienen con el método de Newton, usado tal cual.⁵⁰² Su variante, SFN [*Saddle-Free Newton method*] es un método de optimización basado en regiones de confianza [*trust region*], que limita el tamaño del salto en cada iteración del algoritmo. En vez de utilizar la curvatura de la función en una expansión en serie de Taylor de segundo orden, utiliza la curvatura para

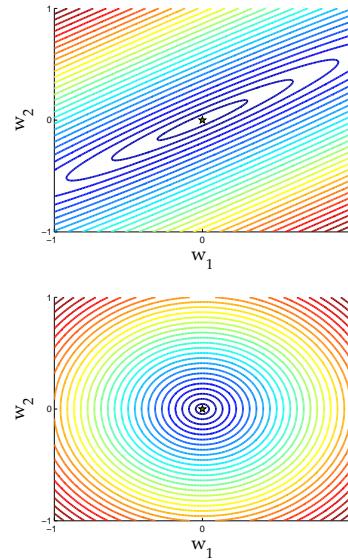


Figura 147: El antes y el después de la superficie de error al multiplicar por la inversa de la matriz hessiana en el método de Newton.

⁵⁰² Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, y Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger, editores, *NIPS'2014 Advances in Neural Information Processing Systems 27*, pages 2933–2941. Curran Associates, Inc., 2014b. URL <https://goo.gl/dE72X5>

delimitar la región de confianza, en la que nos moveremos de acuerdo al gradiente. SFN es equivalente al método de Newton cuando la hessiana es definida positiva (en el entorno de un mínimo) pero, a la vez, es capaz de moverse más rápido en direcciones de baja curvatura, con lo que puede escapar de los puntos de silla de la función.

En cualquier caso, el principal problema del método de Newton es su necesidad de almacenar la matriz hessiana... y calcular su inversa. El cálculo de la inversa se puede evitar si planteamos el problema como un sistema de ecuaciones lineales, del que se obtiene Δx como solución:

$$[H_f(x_n)]\Delta x = -\nabla_x f(x_n)$$

Otros métodos ni siquiera llegan a calcular la matriz hessiana y se conforman trabajando con sustitutos de ésta. Suele tratarse de métodos de tipo iterativo que intentan resolver de forma aproximada la ecuación del método de Newton, motivo por el que se conocen con el nombre de métodos de Newton truncados [*truncated-Newton methods*]⁵⁰³ o métodos libres de hessianas [*Hessian-free optimization*].⁵⁰⁴ Los métodos libres de hessianas, HF, se basan en dos ideas sencillas:

- Dado un vector n-dimensional d , el cálculo de la curvatura de la función a lo largo de la dirección d , Hd , puede realizarse de forma simple usando diferencias finitas mediante una evaluación adicional del gradiente de la función:

$$Hd \approx \frac{\nabla f(x + \epsilon d) - \nabla f(x)}{\epsilon}$$

- Existen algoritmos, como los basados en gradientes conjugados, que sólo requieren realizar multiplicaciones de una matriz por un vector. Estos métodos requieren, formalmente, n iteraciones para garantizar su convergencia (n productos matriz-vector), por lo que podrían parecer tan poco prácticos como calcular directamente la matriz hessiana. Afortunadamente, su comportamiento es tal que progresan significativamente hacia el mínimo tras un número reducido de iteraciones. Como podemos detener la ejecución del algoritmo tras un número limitado de iteraciones, reciben el calificativo de métodos truncados.

El método de Gauss-Newton

El método de Gauss-Newton, nombrado en honor de Carl Friedrich Gauss e Isaac Newton, se utiliza para resolver problemas no lineales de mínimos cuadrados. Es una modificación del método de optimización de Newton realizada por Gauss para evitar el cálculo de la matriz hessiana.

El método de Gauss-Newton aproxima la matriz hessiana utilizando únicamente información proveniente de las primeras derivadas. ¿Cómo?

⁵⁰³ Stephen G. Nash. A survey of truncated-Newton methods. *Journal of Computational and Applied Mathematics*, 124(1):45 – 59, 2000. ISSN 0377-0427. DOI: 10.1016/S0377-0427(00)00426-X. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations

⁵⁰⁴ James Martens. Deep Learning via Hessian-free Optimization. En *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 735–742. Omnipress, 2010. ISBN 978-1-60558-907-7. URL <http://icml2010.haifa.il.ibm.com/papers/458.pdf>

Minimizando el error cuadrático. Si partimos de una función de error de la forma $f(x) = \frac{1}{2} \sum f_k^2$, su gradiente será de la forma

$$g_i = \frac{\partial f}{\partial x_i} = \sum_k f_k \frac{\partial f_k}{\partial x_i}$$

Los elementos H_{ij} de la matriz hessiana se pueden calcular diferenciando esos gradientes g_i con respecto a x_j :

$$H_{ij} = \frac{\partial g_i}{\partial x_j} = \sum_k \left(\frac{\partial f_k}{\partial x_i} \frac{\partial f_k}{\partial x_j} + f_k \frac{\partial^2 f_k}{\partial x_i \partial x_j} \right)$$

Ignorando las segundas derivadas (el segundo término de cada elemento de la sumatoria), se puede aproximar la matriz hessiana como:

$$H_{ij} \approx \sum_k \frac{\partial f_k}{\partial x_i} \frac{\partial f_k}{\partial x_j} = \sum_k J_{ki} J_{kj}$$

donde J_{ki} y J_{kj} son entradas de la matriz jacobiana J .

Usando notación vectorial, podemos aproximar la matriz hessiana usando la matriz jacobiana

$$H_f \approx J_f^\top J_f$$

y representar el gradiente como

$$\nabla f = g = J_f^\top f$$

Si utilizamos las expresiones anteriores en la ecuación del método de optimización de Newton, $x_{n+1} = x_n - H^{-1}g$, obtenemos la fórmula utilizada por el método de Gauss-Newton:

$$x_{n+1} = x_n - (J_f(x_n)^\top J_f(x_n))^{-1} J_f(x_n)^\top f(x_n)$$

donde $J_f(x_n)$ es la matriz jacobiana de la función f evaluada en x_n .

En la práctica, como sucede con el método de Newton, no se calcula explícitamente la inversa de la matriz que aparece en la fórmula anterior, $(J_f(x_n)^\top J_f(x_n))^{-1}$, sino que se actualiza el vector de parámetros x utilizando

$$x_{n+1} = x_n + \Delta x$$

donde Δx se obtiene de resolver el sistema de ecuaciones lineales dado por

$$[J_f(x_n)^\top J_f(x_n)]\Delta x = -J_f(x_n)^\top f(x_n)$$

El método de Gauss-Newton nos permite encontrar el mínimo de una función de forma iterativa. Dada una estimación inicial del vector de parámetros de la función, x_0 , el método va estimando el valor de los parámetros x que minimizan la función f por medio de la resolución de una serie de sistemas de ecuaciones lineales.

Esta función de error se define fácilmente a partir de los errores cometidos sobre cada ejemplo (x_k, y_k) del conjunto de entrenamiento.

Cuando partimos de un punto lejano a la solución, el término que hemos ignorado en nuestra aproximación puede no ser despreciable, lo que hará que la aproximación de la matriz hessiana puede que no resulte demasiado buena y la convergencia del algoritmo sea lenta.

En definitiva, el método de Gauss-Newton nos proporciona una forma más de resolver un problema de optimización... pero sólo funciona si la función de error que queremos minimizar es el error cuadrático (SSE o MSE). Se trata sólo de un caso particular del método de Newton. A diferencia del método de Newton o, ya puestos, del gradiente descendente, el método de Gauss-Newton no lo podemos utilizar para cualquier función de error que sea diferenciable, sino únicamente para minimizar el error cuadrático.

El método de Levenberg–Marquardt

El método de Levenberg–Marquardt⁵⁰⁵ es similar al de Gauss-Newton, en el sentido de que sólo sirve para resolver problemas de mínimos cuadrados, en los que deseemos minimizar una función de error cuadrática. Fue propuesto en 1944 por Kenneth Levenberg,⁵⁰⁶ del ejército de Estados Unidos, y redescubierto en 1963 por Donald Marquardt,⁵⁰⁷ que trabajaba para DuPont. También se conoce con el nombre de mínimos cuadrados amortiguados [DLS: *Damped Least Squares*].

Como en el método de Gauss-Newton, deseamos minimizar una función de error de la forma $f(x) = \frac{1}{2} \sum f_k^2$. Igual que antes, podemos calcular la matriz hessiana a partir de los errores observados en el conjunto de entrenamiento

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_k \left(\frac{\partial f_k}{\partial x_i} \frac{\partial f_k}{\partial x_j} + f_k \frac{\partial^2 f_k}{\partial x_i \partial x_j} \right)$$

En notación vectorial,

$$H_f = J_f^\top J_f + S_f$$

donde S_f es una matriz que recoge las aportaciones de las segundas derivadas, $S_f = \sum_k f_k \nabla^2 f_k$.

Cerca del mínimo de la función de error, los elementos de la matriz S toman valores pequeños, por lo que podemos aproximar la matriz hessiana como antes

$$H_f \approx J_f^\top J_f$$

Hasta ahora, todo se hace exactamente igual que en el método de Gauss-Newton. Sustituyendo la matriz hessiana y el gradiente de la ecuación de Newton por sus aproximaciones basadas en el jacobiano, obtenemos la expresión que ya utilizamos anteriormente:

$$\Delta x = -[J_f(x_n)^\top J_f(x_n)]^{-1} J_f(x_n)^\top f(x_n)$$

⁵⁰⁵ Jorge J. Moré. The Levenberg–Marquardt algorithm: Implementation and theory. En G. A. Watson, editor, *Numerical Analysis: Proceedings of the Biennial Conference Held at Dundee, June 28–July 1, 1977*, volume 630 of *Lecture Notes in Mathematics*, pages 105–116. Springer, 1977. ISBN 978-3-540-35972-2. DOI: 10.1007/BFb0067700

⁵⁰⁶ Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944. ISSN 0033-569X. DOI: 10.1090/qam/10666. URL <http://www.jstor.org/stable/43633451>

⁵⁰⁷ Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963. DOI: 10.1137/0111030. URL <http://www.jstor.org/stable/2098941>

Ahora bien, la expresión anterior requiere invertir una aproximación de la matriz hessiana. Sin embargo, si la matriz hessiana no es definida positiva (esto es, todos sus autovalores o *eigenvalues* no son positivos), como sucede en los puntos de silla, el método de Newton puede realizar actualizaciones en dirección, no a un mínimo, sino a un punto crítico de otro tipo. Además, puede que la matriz hessiana sea singular, por lo que ni siquiera podamos invertirla. Ambos problemas los podemos resolver regularizando la matriz hessiana (o, más bien, la aproximación que estamos utilizando):

$$H_f \approx J_f^\top J_f + \lambda I$$

donde I es la matriz identidad y λ es un factor de regularización.

Por tanto, la fórmula que utilizaremos en el algoritmo iterativo de Levenberg-Marquardt será la siguiente:

$$\Delta x = -[J_f(x_n)^\top J_f(x_n) + \lambda_n I]^{-1} J_f(x_n)^\top f(x_n)$$

donde λ_n puede ir variando a lo largo del proceso de optimización (usando un valor bajo en las etapas iniciales y un valor más elevado en las etapas finales, para prevenir oscilaciones no deseadas y afinar los valores de las variables que minimizan la función).

Si nos fijamos detenidamente en el impacto que tiene el factor de regularización λ sobre el algoritmo de optimización, podemos observar lo siguiente:

- Cuando el factor de regularización es relativamente pequeño, el método de Levenberg-Marquardt se comporta como el método de Newton (o, para ser más precisos, como el método de Gauss-Newton, ya que estamos aproximando la matriz hessiana usando el jacobiano de la función de error).
- Cuando el factor de regularización es grande y la matriz hessiana es pequeña, las iteraciones del algoritmo de Levenberg-Marquardt equivalen a utilizar el gradiente descendente con una tasa de aprendizaje $\eta = 1/\lambda$:

$$\Delta x \approx -[\lambda_n I]^{-1} J_f(x_n)^\top f(x_n) = -\frac{1}{\lambda_n} \nabla_x f(x_n)$$

Es decir, el algoritmo de Levenberg-Marquardt puede verse como un método de optimización a medio camino entre el método de Newton y el gradiente descendente.

Si comenzamos con un valor pequeño para el factor de regularización, rápidamente nos moveremos en dirección hacia el mínimo (como en el método de Newton). Más adelante, podemos aumentar el factor de regularización para que el algoritmo funcione como el gradiente descendente,

Igual que en el gradiente descendente comenzamos utilizando una tasa de aprendizaje inicial que luego vamos reduciendo, en el algoritmo de Levenberg-Marquardt partimos de un factor de regularización pequeño que luego podemos aumentar, al ser éste inversamente proporcional a la tasa de aprendizaje del gradiente descendente: $\eta = 1/\lambda$.

más lento a la hora de converger pero más fiable en situaciones en las que la matriz hessiana no proporciona información útil para localizar el punto donde se halla el mínimo de la función que deseamos minimizar. Conforme aumentamos el valor del parámetro de regularización, el término correspondiente a la diagonal λI domina la aproximación de la matriz hessiana: la dirección elegida por el algoritmo de Levenberg-Marquardt va desviándose de la que se elegiría con el método de Gauss-Newton y va convergiendo con la del gradiente descendente.

Así pues, podemos interpretar el método de Levenberg-Marquardt como una interpolación entre el método de Gauss-Newton y el gradiente descendente. Esto le permite ser más robusto que el método de Gauss-Newton y ser capaz de encontrar una solución aunque partamos muy lejos del mínimo, si bien es cierto que suele ser algo más lento que el método de Gauss-Newton.

El método de Levenberg-Marquardt también puede interpretarse como una modificación del método de Gauss-Newton basado en regiones de confianza: el método se basa en que la aproximación de la matriz Hessiana es válida sólo dentro de una región de confianza de pequeño radio. En vez de resolver $H\Delta x = -g$, el algoritmo de Levenberg-Marquardt resuelve $(H + \lambda I)\Delta x = -g$, donde el parámetro de regularización λ controla el tamaño de la región de confianza. Geométricamente, la modificación equivale a añadir un paraboloide centrado en $\Delta x = 0$, lo que restringe el paso dado en cada iteración del algoritmo.

El truco es cómo ajustar el tamaño de la región de confianza (el parámetro λ). En cada iteración, el ajuste cuadrático amortiguado predice una reducción en la función de coste, Δf_{pred} , que esperamos que sea menor que la real, calculada a partir de Δx como $\Delta f_{real} = f(x + \Delta x) - f(x)$. Evaluando la proporción $\Delta f_{pred}/\Delta f_{real}$ podemos ajustar el tamaño de la región de confianza. Esperamos que esa proporción esté, digamos, entre 0.25 y 0.5. Si está por encima de 0.5, estamos amortiguando demasiado los pasos, por lo que podemos expandir la región de confianza (reducir λ). Si está por debajo de 0.25, no estamos aproximando bien la función real en la región de confianza, por lo que hemos de contraerla (aumentar λ).

Antes de terminar, un pequeño recordatorio: tanto el método de Gauss-Newton como el de Levenberg-Marquardt sólo se pueden utilizar para minimizar el error cuadrático. Si pretende usar otras funciones de error, ya sean funciones regularizadas o la entropía cruzada usada habitualmente en problemas de clasificación, estos dos métodos no resultan aplicables.

En cuanto a sus requisitos computacionales, los métodos de Gauss-Newton y Levenberg-Marquardt son similares. Ambos requieren el cálculo del jacobiano (las derivadas de la función de error con respecto a los parámetros del modelo para cada ejemplo del conjunto de entrenamiento) y, en principio, realizar la inversión de una matriz de tamaño $n \times n$ en cada iteración. Aunque esa inversión se sustituye en la práctica por

Los métodos de optimización basados en regiones de confianza delimitan una región en la que la función objetivo se puede aproximar por un modelo (a menudo cuadrático). Si el modelo es adecuado, la región se puede ampliar. Si la aproximación no es buena, la región se contrae.

Danny C. Sorensen. Newton's method with a model trust region modification. *SIAM Journal on Numerical Analysis*, 19(2):409–426, 1982. doi: 10.1137/0719026

un sistema de ecuaciones lineales, este paso sigue teniendo, en el peor caso, una complejidad computacional de orden cúbico, $O(n^3)$. Por este motivo, se han propuesto algunos métodos alternativos que pretenden conservar las ventajas del método de Newton pero sin las limitaciones computacionales que éste impone.

Métodos quasi-Newton

Los métodos quasi-Newton son métodos iterativos de optimización que se utilizan cuando la matriz hessiana resulta demasiado costosa de calcular en cada iteración del algoritmo. El método de Newton utiliza la matriz hessiana H completa y realiza actualizaciones de la forma $\Delta x = -H^{-1}g$. Los métodos quasi-Newton emplean aproximaciones de la matriz hessiana (o, directamente, de su inversa). Estas aproximaciones se calculan directamente a partir de los cambios observados en el gradiente de la función de error, coste o pérdida que pretendemos minimizar. Es decir, evaluaciones sucesivas del gradiente de la función nos permiten ir aproximando el valor de la matriz hessiana (o su inversa).

Los métodos quasi-Newton pueden considerarse extensiones del método de la secante para problemas multidimensionales. El método de la secante es un método iterativo para encontrar los ceros o raíces de una función f que, cuando lo empleamos para encontrar los ceros de la derivada de la función, nos permite encontrar sus puntos críticos. Es una variante del método de Newton-Raphson que se basa en aproximar la derivada de la función obteniendo la pendiente de la secante, la recta que pasa por los puntos $(x_n, f(x_n))$ y $(x_{n-1}, f(x_{n-1}))$. Matemáticamente, el método de la secante emplea la siguiente recurrencia:

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

Su orden de convergencia viene dado por la razón áurea, $\varphi = (1 + \sqrt{5})/2$, que, aun siendo superlineal, es inferior al método de Newton-Raphson, que utiliza la derivada de la función y tiene un orden de convergencia cuadrático. Su fórmula le sonará muy familiar, al ser la base del método de optimización de Newton:

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

En el caso multidimensional, la ecuación derivada del método de la secante está infradeterminada (menos ecuaciones que variables desconocidas), por lo que existen distintos métodos quasi-Newton en función de cómo se imponen restricciones a la solución.

Como en el método de Newton, utilizamos una aproximación en serie de Taylor de la función:

$$f(x_k + \Delta x) \approx f(x_k + \Delta x) + \nabla f(x_k)^\top \Delta x + \frac{1}{2} \Delta x^\top B \Delta x$$

donde B será una aproximación de la matriz hessiana en x_k , $B \approx H_f(x_0)$. El gradiente de esta aproximación con respecto a Δx es

$$\nabla f(x_k + \Delta x) \approx \nabla f(x_k) + B\Delta x$$

Igualando a cero el gradiente, obtenemos la ecuación utilizada por el método de Newton:

$$\Delta x = -B^{-1}\nabla f(x_k)$$

Ahora bien, la aproximación B de la matriz hessiana la elegimos de tal forma que se verifique la ecuación de la secante:

$$\nabla f(x_k + \Delta x) = \nabla f(x_k) + B\Delta x$$

Esta ecuación está infradeterminada. En una dimensión, equivale al método de la secante. En múltiples dimensiones, los diferentes métodos quasi-Newton difieren en la solución que eligen de esta ecuación. La mayoría de los métodos escogen soluciones que sean simétricas ($B = B^\top$). Además, suelen escoger una estimación B_{k+1} que esté lo más cerca posible de B_k , minimizando algún tipo de norma: $B_{k+1} = \arg \min_B \|B - B_k\|_V$. Un valor inicial $B_0 = I * x$ suele ser válido, siempre que B_0 sea una matriz definida positiva (si lo que buscamos es un mínimo de la función).

El primer método quasi-Newton fue propuesto en 1959 por el físico William C. Davison, del Argonne National Laboratory en las afueras de Chicago, Illinois. Su método, popularizado por Roger Fletcher y Michael J.D. Powell en 1963, se conoce como fórmula DFP [Davison-Fletcher-Powell]. Otros métodos quasi-Newton más populares en la actualidad son los métodos SR1 [Symmetric Rank 1], BFGS [Broyden-Fletcher-Goldfarb-Shanno], BHHH (el algoritmo de Berndt-Hall-Hall-Hausman, similar al de Gauss-Newton) y la familia de métodos de Broyden (combinaciones lineales de DFP y BGFS: $(1 - \varphi_k)B_{k+1}^{BFGS} + \varphi_k B_{k+1}^{DFP}$, con $\varphi \in [0, 1]$).

Una de las principales ventajas de los métodos quasi-Newton con respecto al método de Newton (o sus variantes de Gauss-Newton y Levenberg-Marquardt) es que no hace falta invertir su aproximación B de la matriz hessiana, ya que son capaces de generar una estimación de B^{-1} directamente. Para ello, calculan la aproximación de la matriz inversa $M_{k+1} = M_{k+1}^{-1}$ usando la fórmula de Sherman-Morrison, que permite calcular la inversa de la suma de una matriz invertible A y el producto tensorial de dos vectores uv^\top como:

$$(A + uv^\top)^{-1} = A^{-1} - \frac{A^{-1}uv^\top A^{-1}}{1 + v^\top A^{-1}u}$$

Si definimos $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, éstas son las fórmulas que podemos utilizar directamente en distintos métodos quasi-Newton de optimización:

- DFP [Davison-Fletcher-Powell]:⁵⁰⁸

$$M_{k+1} = B_{k+1}^{-1} = M_k + \frac{\Delta x_k \Delta x_k^\top}{\Delta x_k^\top y_k} - \frac{M_k y_k y_k^\top M_k}{y_k^\top M_k y_k}$$

- BFGS [Broyden–Fletcher–Goldfarb–Shanno], dual de DFP:⁵⁰⁹

$$M_{k+1} = B_{k+1}^{-1} = \left(I - \frac{\Delta x_k y_k^\top}{y_k^\top \Delta x_k} \right) M_k \left(I - \frac{y_k \Delta x_k^\top}{y_k^\top \Delta x_k} \right) + \frac{\Delta x_k \Delta x_k^\top}{y_k^\top \Delta x_k}$$

- SR1 [Symmetric Rank One]:⁵¹⁰

$$M_{k+1} = B_{k+1}^{-1} = M_k + \frac{(\Delta x_k - M_k y_k)(\Delta x_k - M_k y_k)^\top}{(\Delta x_k - M_k y_k)^\top y_k}$$

- Método de Broyden:⁵¹¹

$$M_{k+1} = B_{k+1}^{-1} = M_k + \frac{(\Delta x_k - M_k y_k)\Delta x_k^\top M_k}{\Delta x_k^\top M_k y_k}$$

En cada iteración de un algoritmo quasi-Newton, el producto $M_k \nabla f(x_k)$ nos indica la dirección en la que hemos de dar el salto. La magnitud de ese salto se determina realizando una búsqueda lineal en esa dirección. Un algoritmo quasi-Newton, por tanto, realizaría los siguientes pasos:

- Se selecciona un vector solución inicial x_0 y una primera aproximación de la inversa de la matriz hessiana, p.ej. $M_0 = I$ (asumiendo que partimos de una matriz hessiana inicial $B_0 = I$).
- Se calcula el gradiente de la función objetivo $\nabla f(x_k)$.
- Se determina la dirección en la que daremos un salto: $d_k = -M_k \nabla f(x_k)$.
- Se realiza una búsqueda lineal para determinar $x_{k+1} = x_k + \alpha_k d_k$, donde $\alpha_k = \arg \min_{\alpha \geq 0} f(x_k + \alpha d_k)$.
- Se calculan las diferencias $\Delta x_k = x_{k+1} - x_k$ e $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$.
- Se actualiza nuestra estimación de la inversa de la matriz hessiana, p.ej. en el algoritmo BFGS:

$$M_{k+1} = B_{k+1}^{-1} = \left(I - \frac{\Delta x_k y_k^\top}{y_k^\top \Delta x_k} \right) M_k \left(I - \frac{y_k \Delta x_k^\top}{y_k^\top \Delta x_k} \right) + \frac{\Delta x_k \Delta x_k^\top}{y_k^\top \Delta x_k}$$

- Mientras x_k no converja, se vuelve al segundo paso y se ejecuta una iteración más del algoritmo.

⁵⁰⁸ William C. Davidon. Variable metric method for minimization. *SIAM Journal on Optimization*, 1(1):1–17, 1991. DOI: 10.1137/0801001

⁵⁰⁹ Charles George Broyden. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970. ISSN 0272-4960. DOI: 10.1093/imamat/6.1.76

⁵¹⁰ Andrew R. Conn, Nicholas I.M. Gould, y Philippe L. Toint. Convergence of quasi-Newton matrices generated by the symmetric rank one update. *Mathematical Programming*, 50(1):177–195, 1991. ISSN 1436-4646. DOI: 10.1007/BF01594934

⁵¹¹ Charles George Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation*, 19(92):577–593, 1965. ISSN 0025-5718. DOI: 10.1090/S0025-5718-1965-0198670-6. URL <http://www.jstor.org/stable/2003941>

Variantes de BFGS

El algoritmo BFGS es el más popular de los algoritmos quasi-Newton. Recibe su nombre de Charles George Broyden (Universidad de Essex, Inglaterra), Roger Fletcher (Universidad de Dundee, Escocia), Donald Goldfarb (Universidad de Columbia, Nueva York) y David Shanno (Universidad de Rutgers, Nueva Jersey).

BFGS modifica la aproximación de la matriz hessiana realizando una actualización de rango 2, como el método DFS anterior, del que es dual. En cambio, el método de Broyden y el algoritmo SR1 recurren a actualizaciones de rango 1. Al realizar una actualización de rango 2, la fórmula usada por el método quasi-Newton asegura que las aproximaciones realizadas sean siempre simétricas y definidas positivas.

En realidad, BFGS se basa en añadir a B_k dos matrices simétricas de rango 1, tales que su suma es una matriz de rango 2:

$$B_{k+1} = B_k + U_k + V_k$$

Las actualizaciones que se realizan son de la forma:

$$B_{k+1} = B_k + \alpha uu^\top + \beta vv^\top$$

donde se usa $u = y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ y $v = B_k \Delta x_k$.

Al imponer la condición de la secante, $B_{k+1}\Delta x = y_k$, se obtienen los valores de α y β :

$$\begin{aligned}\alpha &= \frac{1}{y_k^\top \Delta x_k} \\ \beta &= -\frac{1}{\Delta x_k^\top B_k \Delta x_k}\end{aligned}$$

De ahí sale la fórmula de actualización del algoritmo BFGS:

$$B_{k+1} = B_k + \frac{y_k - B_k \Delta x_k}{\Delta x_k^\top \Delta x_k} \Delta x_k^\top$$

Como para resolver un problema de optimización lo que realmente nos interesa es la inversa de la aproximación de la matriz hessiana, utilizamos la fórmula de Sherman-Morrison y llegamos a la expresión que mostramos en la sección anterior:

$$M_{k+1} = B_{k+1}^{-1} = \left(I - \frac{\Delta x_k y_k^\top}{y_k^\top \Delta x_k} \right) M_k \left(I - \frac{y_k \Delta x_k^\top}{y_k^\top \Delta x_k} \right) + \frac{\Delta x_k \Delta x_k^\top}{y_k^\top \Delta x_k}$$

La expresión anterior, que puede parecer compleja, se puede calcular de forma eficiente, sin necesidad de mantener matrices auxiliares en memoria, si tenemos en cuenta que M_k es siempre simétrica y que los términos $y_k^\top M_k y_k$ y $\Delta x_k^\top y_k$ son escalares. Esto nos permite reescribir la expresión anterior como:

$$M_{k+1} = B_{k+1}^{-1} = M_k + \frac{\Delta x_k^\top y_k + y_k^\top M_k y_k}{(\Delta x_k^\top y_k)^2} (\Delta x_k \Delta x_k^\top) - \frac{1}{\Delta x_k^\top y_k} (M_k y_k \Delta x_k^\top - \Delta x_k y_k^\top M_k)$$

Intercambiando $H \leftrightarrow H^{-1}$ y $\Delta x_k \leftrightarrow y_k$ pasamos de DFS a BFGS, de ahí que se consideren duales.

El rango de una matriz es su número de filas (o columnas) linealmente independientes.

Aun así, seguimos necesitando almacenar en memoria dos matrices de tamaño $n \times n$, las matrices M_k y M_{k+1} , lo que hace que el algoritmo no sea realmente escalable al requerir un espacio cuadrático con respecto al número de parámetros del problema de optimización. Por este motivo se han diseñado aproximaciones del método BFGS que utilizan una cantidad de memoria limitada.

La más conocida de las variantes del método BFGS es el algoritmo BFGS de memoria limitada [*limited-memory BFGS*], a menudo abreviado como L-BFGS o LM-BFGS. Como el algoritmo del que proviene, L-BFGS estima la inversa de la matriz hessiana para guiar su búsqueda hasta el mínimo de la función de coste o pérdida. Ahora bien, en lugar de utilizar una aproximación densa de la hessiana inversa, usando una matriz de tamaño $n \times n$, el algoritmo L-BFGS almacena sólo algunos vectores para representar esa aproximación. De esta forma, sólo requiere un espacio lineal en memoria y resulta adecuado para problemas de optimización con un número elevado de variables.

La estrategia utilizada por L-BFGS es mantener en memoria un historial de las últimas m actualizaciones de la posición x_k y del gradiente $\nabla f(x_k)$. El tamaño de ese historial es pequeño (generalmente, $m < 10$) y se emplea donde aparece una multiplicación de la matriz M_k por un vector en el algoritmo BFGS.

Como siempre, partimos de un valor inicial x_0 y realizamos una estimación del gradiente $g_k = \nabla f(x_k)$. L-BFGS cambia con respecto a los demás métodos quasi-Newton en el cálculo de la dirección $d_k = -M_k g_k$. Existen diversas formas de calcular el vector d_k a partir del historial de actualizaciones. Aquí emplearemos uno que utiliza dos bucles [*two loop recursion*].⁵¹²

Suponiendo que tenemos las m últimas actualizaciones de la forma $\Delta x_k = x_{k+1} - x_k$ e $y_k = g_{k+1} - g_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, definimos el escalar $\rho_k = 1/(y_k^\top \Delta x_k)$. La recurrencia utilizada por el algoritmo BFGS para aproximar la inversa de la matriz hessiana es, en términos de estas variables, de la forma:

$$M_{k+1} = (I - \rho_k \Delta x_k y_k^\top) M_k (I - \rho_k y_k \Delta x_k^\top) + \rho_k \Delta x_k \Delta x_k^\top$$

En la iteración k , definimos la secuencia de vectores $q_{k-m}..q_k$ a partir de $q_k = g_k$ usando la recurrencia $q_i = (I - \rho_i y_i \Delta x_i^\top) q_{i+1}$. El algoritmo recursivo dado por la expresión anterior lo podemos convertir fácilmente en un algoritmo iterativo definiendo $\alpha_i = \rho_i \Delta x_i^\top q_{i+1}$ y calculando q_i a partir de q_{i+1} como sigue: $q_i = q_{i+1} - \alpha_i y_i$.

Definimos una segunda secuencia de vectores $z_{k-m}..z_k$ en la que $z_i = M_i q_i$. Otro algoritmo recursivo nos permite calcular esos vectores a partir de $z_{k-m} = M_k^0 q_{k-m}$, para lo que definimos $\beta_i = \rho_i y_i^\top z_i$ y aplicamos la fórmula $z_{i+1} = z_i + (\alpha_i - \beta_i) \Delta x_i$. El valor de z_k nos da la dirección en la que deberíamos realizar la búsqueda hacia un máximo,

⁵¹² Jorge Nocedal. Updating Quasi-Newton Matrices with Limited Storage. *Mathematics of Computation*, 35(151):773–782, 1980. ISSN 0025-5718. DOI: 10.1090/S0025-5718-1980-0572855-7. URL <https://www.jstor.org/stable/2006193>

por lo que utilizaremos $-z_k$ cuando estemos minimizando.

Sólo nos falta definir la aproximación inicial de la inversa de la matriz hessiana, M_k^0 , de la que partirá el algoritmo L-BFGS. Normalmente se elige una matriz diagonal, para que el cálculo de z sólo requiera una multiplicación elemento a elemento:

$$M_k^0 = \frac{y_{k-m} \Delta x_{k-m}^\top}{y_{k-m}^\top y_{k-m}}$$

A la hora de implementar el algoritmo, partimos de un valor inicial de q , $q_k = g_k$, que vamos actualizando usando la expresión $q_i = q_{i+1} - \alpha_i y_i$. Tras $m - 1$ iteraciones, tendremos en q el valor de q_{k-m} . Ese valor lo empleamos para inicializar el valor de z : $z = z_{k-m} = M_k^0 q_{k-m}$. Tras otras $m - 1$ iteraciones en las que recurrimos a la expresión $z_{i+1} = z_i + (\alpha_i - \beta_i) \Delta x_i$, obtendremos el valor de la dirección que emplearemos en la fase búsqueda lineal del algoritmo quasi-Newton (sin olvidarnos de cambiar el signo de z para minimizar).

Por tanto, el algoritmo L-BFGS no necesita almacenar ninguna matriz densa. Nos basta con almacenar $2m$ vectores con el historial de actualizaciones Δx_k e y_k , así como mantener en memoria tres vectores adicionales para q , z y la diagonal de M_k^0 .

En cuanto a la fase de búsqueda lineal, L-BFGS no requiere realizar una búsqueda exacta (p.ej. el método de Brent). Nos sirve cualquier algoritmo aproximado, como la búsqueda lineal con *backtracking*.⁵¹³

La búsqueda lineal con *backtracking* es un método de búsqueda lineal que comienza con una estimación relativamente grande del tamaño del salto e iterativamente lo va reduciendo [*backtracking*] hasta que se observe una disminución en la función objetivo que corresponda con la reducción esperada dado el gradiente de la función. Nuestro objetivo es encontrar un valor de α que nos ayude a reducir el valor de la función en la dirección $-z$, de forma que $f(x - \alpha z)$ sea menor que $f(x)$ sin emplear demasiados recursos computacionales. No nos importa no calcular al valor óptimo α^* que minimiza $f(x - \alpha z)$, nos basta con lograr una mejora razonable.

Partiendo de un tamaño de paso máximo $\alpha_0 > 0$, calculamos la pendiente de la función en la dirección de búsqueda $-z$ como $m = -z^\top \nabla f(x)$, lo que dará un valor negativo ($m < 0$). Usando un parámetro de control $c \in (0, 1)$, utilizamos la condición de Armijo-Goldstein para evaluar si un movimiento de la posición x a la posición $(x - \alpha z)$ nos proporciona una mejora adecuada en nuestra función objetivo: $f(x - \alpha z) \leq f(x) + c \alpha m$. Como nuestra estimación inicial α_0 será elevada y, seguramente, no satisfará la condición de Armijo-Goldstein. Mientras no se satisfaga la condición, iremos reduciendo el valor de α usando un factor $\tau \in (0, 1)$: $\alpha_j = \tau \alpha_{j-1}$. En cuanto encontremos un valor de α_j que satisfaga la condición, lo devolvemos como resultado.

En resumen, este algoritmo de búsqueda lineal se limita a ir reduciendo

⁵¹³ Larry Armijo. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics*, 16(1):1–3, 1966. ISSN 0030-8730. URL <https://msp.org/pjm/1966/16-1/p01.xhtml>

un valor inicial α_0 por un factor τ hasta satisfacer la condición de Armijo-Goldstein. En su artículo de 1966, Armijo propuso utilizar $c = \tau = 1/2$. El algoritmo siempre terminará en un número finito de pasos para cualesquiera valores positivos de c y τ menores que 1, ya que cualquier valor de α lo suficientemente pequeño satisfará siempre la condición de Armijo-Goldstein que se utiliza como criterio de parada.

L-BFGS no es la única variante de BFGS que se ha propuesto. De hecho, existen múltiples variantes de L-BFGS para diferentes situaciones, como L-BFGS-B,⁵¹⁴ diseñada por Richard Byrd para incorporar restricciones simples del tipo $l_i < x_i < u_i$ en el algoritmo L-BFGS propuesto por Jorge Nocedal. Esas restricciones se denominan *box* o *bounding constraints* en inglés, de ahí la B final del nombre del algoritmo. Incluso hay variantes *online* de L-BFGS, como O-LBFGS⁵¹⁵ o SFO [Sum of Functions Optimizer],⁵¹⁶ que permiten evaluar la función de error y su gradiente sobre muestras aleatorias del conjunto de datos en cada iteración, de forma similar al gradiente descendente estocástico.

Los métodos L-BFGS se diferencian de BFGS en que no almacenan la aproximación completa de la matriz hessiana, lo que los hace escalables. Si llevamos esta estrategia al extremo, llegamos a los algoritmos BFGS sin memoria [*memoryless BFGS*]. En estos algoritmos no necesitamos recordar nada de una iteración a otra, ni una aproximación completa de H^{-1} como en BFGS ni un conjunto de vectores como en el historial de L-BFGS.

El método de la secante en un paso, OSS [*One-Step Secant*],^{517,518} es un ejemplo de algoritmo BFGS sin memoria. Utiliza la misma estrategia que en L-BFGS pero, en vez de almacenar información acerca del historial de actualizaciones, comienza cada iteración con la suposición de que la aproximación anterior de la matriz hessiana era siempre la matriz identidad. Si establecemos $M_k = I$ en la fórmula de BFGS, obtenemos

$$M_{k+1} = B_{k+1}^{-1} = \left(I - \frac{\Delta x_k y_k^\top}{y_k^\top \Delta x_k} \right) \left(I - \frac{y_k \Delta x_k^\top}{y_k^\top \Delta x_k} \right) + \frac{\Delta x_k \Delta x_k^\top}{y_k^\top \Delta x_k}$$

de donde se puede derivar la dirección de búsqueda

$$d_{k+1} = -g_{k+1} + C_k y_k + D_k \Delta x_k$$

siendo

$$C_k = \frac{\Delta x_k^\top g_{k+1}}{y_k^\top \Delta x_k}$$

y

$$D_k = - \left(1 + \frac{y_k^\top y_k}{y_k^\top \Delta x_k} \right) C_k + \frac{y_k^\top g_{k+1}}{y_k^\top \Delta x_k}$$

Aunque su coste computacional es ligeramente inferior a L-BFGS, su convergencia es más lenta que la de los métodos anteriores, ya que estamos

⁵¹⁴ Richard H. Byrd, Peihuang Lu, Jorge Nocedal, y Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995. DOI: 10.1137/0916069

⁵¹⁵ Nicol N. Schraudolph, Jin Yu, y Simon Günter. A Stochastic Quasi-Newton Method for Online Convex Optimization. En Marina Meila y Xiaotong Shen, editores, *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, volume 2 of *Proceedings of Machine Learning Research*, pages 436–443, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR. URL <http://proceedings.mlr.press/v2/schraudolph07a.html>

⁵¹⁶ Jascha Sohl-Dickstein, Ben Poole, y Surya Ganguli. Fast large-scale optimization by unifying stochastic gradient and quasi-Newton methods. En Eric P. Xing y Tony Jebara, editores, *Proceedings of the 31st International Conference on Machine Learning*, volume 32(2) of *Proceedings of Machine Learning Research*, pages 604–612, Beijing, China, 22–24 Jun 2014. PMLR. URL <http://proceedings.mlr.press/v32/sohl-dicksteinb14.html>

⁵¹⁷ Roberto Battiti. First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method. *Neural Computation*, 4(2):141–166, 1992. DOI: 10.1162/neco.1992.4.2.141

⁵¹⁸ Roberto Battiti y Giampietro Tocchioli. Learning with first, second, and no derivatives: A case study in high energy physics. *Neurocomputing*, 6(2):181 – 206, 1994. ISSN 0925-2312. DOI: 10.1016/0925-2312(94)90054-X

aprovechando menos información acerca de las segundas derivadas de la función de coste.

Cuando se utiliza un algoritmo exacto de búsqueda lineal, las direcciones generadas por OSS son mutuamente conjugadas, como en el siguiente tipo de técnicas de optimización que analizaremos: los métodos basados en el uso de gradientes conjugados. Frente a ellos, las variantes de BFGS (como L-BFGS u OSS) tienen la ventaja de que siguen funcionando bien cuando la búsqueda lineal se realiza sólo de forma aproximada, como vimos anteriormente. Incluso se han llegado a proponer métodos quasi-Newton en los que se llega a prescindir por completo de la búsqueda lineal [*line-search-free*].⁵¹⁹

Gradientes conjugados

El método de Newton no resulta adecuado para problemas de optimización con muchas variables, como tampoco los métodos quasi-Newton que requieren el almacenamiento de una matriz completa que aproxime la matriz hessiana (o su inversa). El tamaño de esta matriz, cuadrático con respecto al número de variables del problema, hace que esos algoritmos no resulten escalables y, por tanto, no se empleen en *deep learning*. Las variantes de métodos quasi-Newton más eficientes en el uso de memoria, como L-BFGS u OSS, sí que se pueden emplear con éxito para entrenar redes neuronales artificiales.

Otra alternativa que se encuentra a nuestra disposición consiste en utilizar métodos de Krylov, propuestos por el matemático e ingeniero naval ruso Alexei Krylov en 1931. Los métodos de Krylov son técnicas de tipo iterativo que se pueden emplear para realizar varias operaciones matemáticas, como invertir una matriz de forma aproximada o encontrar aproximaciones de los autovectores [*eigenvectors*] o autovalores [*eigenvalues*] de una matriz, realizando únicamente productos de matrices por vectores, como los algoritmos de Arnoldi o Lanczos. Por ejemplo, si queremos aplicar métodos de Krylov en cálculos que involucren a la matriz hessiana, podemos calcular el producto de la matriz hessiana H y un vector arbitrario v de la siguiente forma:⁵²⁰

$$Hv = \nabla_x [(\nabla_x f(x))v]$$

donde no es necesario trabajar con matrices, sólo con vectores. Además, los cálculos involucrados se pueden automatizar con bibliotecas de diferenciación automática, por lo que ni siquiera resulta necesario que expandamos esa expresión manualmente para aproximar el gradiente usando diferencias finitas.

Cuando nos enfrentamos a un problema de optimización, los métodos de Krylov se han utilizado con éxito en el diseño de técnicas de optimización basadas en el uso de gradientes conjugados,⁵²¹ propuestas

⁵¹⁹ Homayoon S.M. Beigi. Neural network learning through optimally conditioned quadratically convergent methods requiring no line search. En *Proceedings of 36th Midwest Symposium on Circuits and Systems*, pages 109–112 vol.1, Aug 1993. DOI: 10.1109/MWS-CAS.1993.343053

⁵²⁰ Bruce Christianson. Automatic Hessians by reverse accumulation. *IMA Journal of Numerical Analysis*, 12(2):135–150, 1992. DOI: 10.1093/imanum/12.2.135

⁵²¹ Magnus R. Hestenes y Eduard Stiefel. "Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952. DOI: 10.6028/jres.049.044

originalmente en 1952 por Magnus Hestenes y Eduard Stiefel, de la Universidad de California en Los Ángeles, más conocida como UCLA. Desde un punto de vista formal, el método de los gradientes conjugados es un algoritmo numérico de tipo iterativo que permite resolver sistemas de ecuaciones lineales en los que la matriz es simétrica y definida positiva (dos de las propiedades que nos gusta que tenga la matriz hessiana cuando queremos resolver un problema de minimización). Su generalización, basada en el uso de gradientes biconjugados, se puede emplear para casos en los que la matriz no sea simétrica.

El método de los gradientes conjugados nos permite evitar el cálculo de la inversa de la matriz hessiana descendiendo iterativamente a lo largo de direcciones conjugadas. Recordemos que dos direcciones son conjugadas si la optimización realizada en una de ellas no interfiere en la optimización realizada en la otra.

Si realizásemos búsquedas lineales directamente en la dirección que nos proporciona el gradiente de la función de coste o pérdida, nos encontraríamos con que, en sucesivas iteraciones, iríamos realizando una trayectoria en zig-zag, en la que el trabajo de optimización realizado en una iteración del algoritmo se deshace parcialmente en la siguiente.

Si nos movemos siempre en la dirección del gradiente $d_k = -g_k = -\nabla f(x_k)$, cada iteración involucra una búsqueda lineal en la dirección del gradiente, hasta llegar al punto en el que el gradiente en esa dirección sea cero: $\nabla f(x) \cdot d_k = 0$. Como el gradiente en ese punto determinará la dirección en la que nos moveremos en la siguiente iteración, el gradiente d_{k+1} no tendrá contribución alguna en la dirección de d_k . Por tanto, en cada iteración escogeremos siempre una dirección d_{k+1} ortogonal a la dirección anterior d_k . Esta selección de la dirección de búsqueda no preserva el mínimo de acuerdo a la dirección anterior, por lo que nuestro progreso zigzaguea y en cada iteración estamos repitiendo trabajo que ya conseguimos en la iteración anterior.

El método de los gradientes conjugados está diseñado para que las direcciones se vayan seleccionando de forma que no deshagamos trabajo ya hecho. Las direcciones se van escogiendo de acuerdo a la expresión

$$d_k = \nabla f(x) + \beta_k d_{k-1}$$

donde el parámetro β_k controla hasta qué punto debemos seguir moviéndonos en la dirección d_{k-1} para preservar el mínimo de la función de coste en esa dirección.

Matemáticamente, dos direcciones d_k y d_{k-1} son conjugadas si se cumple que $d_k^\top H d_{k-1} = 0$, donde H es la matriz hessiana. Sin embargo, calcular la matriz hessiana para determinar las direcciones conjugadas no nos ofrecería ventajas computacionales con respecto al método de Newton. Por suerte, podemos elegir los valores de β_k sin necesidad de recurrir a la matriz hessiana. A continuación se muestran varias de las

formas que se han propuesto para establecer el valor de β_k :

- Fletcher-Reeves:⁵²²

$$\beta_k^{FR} = \frac{g_k^\top g_k}{g_{k-1}^\top g_{k-1}}$$

- Polak-Ribière:⁵²³

$$\beta_k^{PR} = \frac{g_k^\top (g_k - g_{k-1})}{g_{k-1}^\top g_{k-1}}$$

- Hestenes-Stiefel:⁵²⁴

$$\beta_k^{HS} = \frac{g_k^\top (g_k - g_{k-1})}{d_{k-1}^\top (g_k - g_{k-1})}$$

- Dai-Yuan:⁵²⁵

$$\beta_k^{DY} = \frac{g_k^\top g_k}{d_{k-1}^\top (g_k - g_{k-1})}$$

- Descenso conjugado [*conjugate descent*]:⁵²⁶

$$\beta_k^{CD} = \frac{g_k^\top g_k}{g_{k-1}^\top (x_k - x_{k-1})}$$

Las cinco fórmulas resultan completamente equivalentes en la optimización de funciones cuadráticas. Para problemas de optimización no lineal, la selección de una u otra es más una cuestión de preferencias personales que otra cosa.

En teoría, usando aritmética exacta, el algoritmo de optimización basado en los gradientes descendentes converge a la solución en n pasos. Es lógico si tenemos en cuenta que, dada la forma en la que se seleccionan las direcciones, se asegura que el gradiente con respecto a las direcciones previas nunca aumenta en magnitud (y cada iteración anula el gradiente en una dirección gracias a la búsqueda lineal). No obstante, este resultado formal va acompañado de dos noticias, una buena y una mala. La mala noticia es que, debido a los errores de redondeo que se producen al realizar operaciones en coma flotante en un ordenador digital, puede que se necesiten más de n pasos (o no se llegue a converger). La buena noticia es que, con un poquitín de suerte, se puede conseguir una muy buena aproximación del mínimo en muchas menos de n iteraciones.

Cuando se aplica a la minimización de funciones no cuadráticas, situación a la que se hace referencia con el término gradientes conjugados no lineales [*nonlinear conjugate gradients*], el método de los gradientes conjugados sigue siendo aplicable pero requiere una ligera modificación. Sin una función objetivo cuadrática, las direcciones conjugadas no garantizan que nos mantengamos en el mínimo de la función objetivo con respecto a las direcciones previas, por lo que los métodos de gradientes conjugados no lineales incluyen reinicializaciones ocasionales.⁵²⁷ Podemos reiniciar β_k a cero cada cierto número de iteraciones (por ejemplo, 5)⁵²⁸ o

⁵²² Roger Fletcher y C. M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7(2):149–154, 1964. DOI: 10.1093/comjnl/7.2.149

⁵²³ Elijah Polak y G. Ribière. Note sur la convergence de méthodes de directions conjuguées. *Revue française d'informatic et de recherche opérationnelle. Série rouge*, 3(R1):35–43, 1969. URL http://www.numdam.org/article/M2AN_1969_3_1_35_0.pdf

⁵²⁴ Magnus R. Hestenes y Eduard Stiefel. "Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952. DOI: 10.6028/jres.049.044

⁵²⁵ Y.H. Dai y Y. Yuan. A nonlinear conjugate gradient method with a strong global convergence property. *SIAM Journal on Optimization*, 10(1):177–182, 1999. DOI: 10.1137/S1052623497318992

⁵²⁶ Roger Fletcher. *Practical Methods of Optimization*. Wiley, 2nd edition, 2000. ISBN 0471915475

⁵²⁷ M. J. D. Powell. Restart procedures for the conjugate gradient method. *Mathematical Programming*, 12(1):241–254, Dec 1977. ISSN 1436-4646. DOI: 10.1007/BF01593790

⁵²⁸ P. Patrick van der Smagt. Minimization methods for training feedforward neural networks. *Neural Networks*, 7(1):1 – 11, 1994. ISSN 0893-6080. DOI: 10.1016/0893-6080(94)90052-3

utilizar un criterio del estilo de $\beta_k = \max\{0, \beta_k^{PR}\}$, que reinicializa la dirección automáticamente, olvidándose de la dirección anterior y siguiendo la dirección del gradiente en cuanto $\beta_k < 0$. Con reinicializaciones periódicas, se garantiza la convergencia de los gradientes conjugados no lineales.

El algoritmo de optimización basado en gradientes conjugados no lineales, por tanto, queda como sigue:

- Inicialización:

Se calcula el gradiente $g_0 = -\nabla f(x_0)$ y se realiza una búsqueda lineal en la dirección del gradiente, $\alpha_0 = \arg \min_\alpha f(x_0 + \alpha g_0)$, lo que nos permite movernos hasta $x_1 = x_0 + \alpha_0 g_0$ en la dirección $d_0 = g_0$.

- En cada iteración del algoritmo:

- Se calcula la dirección de máxima pendiente, dada por el gradiente $g_k = -\nabla f(x_k)$.
- Se calcula el parámetro β_k de acuerdo a alguna de las fórmulas (p.ej. Polak-Ribière) y, ocasionalmente, se resetea a cero.
- Se obtiene una dirección conjugada: $d_k = g_k + \beta_k d_{k-1}$.
- Se realiza una búsqueda lineal para obtener $\alpha_k = \arg \min_\alpha f(x_k + \alpha d_k)$.
- Se actualiza la posición: $x_{k+1} = x_k + \alpha_k d_k$.

El método de los gradientes conjugados llega a un compromiso entre la simplicidad del gradiente descendente y la convergencia más rápida de los métodos de segundo orden como el método de Newton. Si reinicializásemos β_k a cero en cada iteración, estaríamos aplicando el algoritmo del gradiente descendente. Sin embargo, el uso de gradientes conjugados nos evitan seguir la típica trayectoria en zig-zag del gradiente descendente, con lo que la convergencia hacia el mínimo se consigue más rápidamente.

En comparación con los métodos quasi-Newton, tipo L-BFGS, los gradientes conjugados suelen requerir un número superior de iteraciones, si bien es cierto que las iteraciones de los métodos quasi-Newton requieren la realización de más cálculos que las de los gradientes conjugados. Esta diferencia en el coste de cada iteración, en la práctica, se compensa parcialmente al realizar la búsqueda lineal: los gradientes conjugados requieren una búsqueda lineal exacta, que requiere más evaluaciones de la función que la búsqueda aproximada con la que funcionan algoritmos como BFGS y sus variantes más escalables.

Tanto las versiones escalables de los métodos quasi-Newton (L-BFGS y OSS) como el método de los gradientes conjugados son lineales en tiempo y espacio, por lo que resultan adecuados para problemas complejos de optimización como los que nos encontramos en *deep learning*, en los que

tenemos que ajustar los millones de parámetros de una red neuronal artificial.

El uso de gradientes conjugados puede interpretarse como una forma inteligente de añadir momentos al gradiente descendente.⁵²⁹ Es como si, automáticamente, estuviésemos eligiendo una tasa de aprendizaje óptima (búsqueda lineal) y un momento adecuado (el parámetro β_k) en cada época del entrenamiento de la red.⁵³⁰ Cuando se utilizan funciones de error cuadráticas, se puede demostrar que el método de los gradientes conjugados es equivalente a un algoritmo basado en el gradiente descendente con momentos que haya sido ajustado de forma óptima.

El método de los gradientes conjugados se ha utilizado tradicionalmente para entrenar redes neuronales utilizando aprendizaje por lotes.⁵³¹ Se han llegado a proponer variantes específicas del método para entrenar redes neuronales, como los gradientes conjugados escalados [*scaled conjugate gradients*].⁵³² Igual que sucedía con L-BFGS, también se han sugerido adaptaciones del método para que se pueda emplear en el aprendizaje con minibatches, mucho más habitual en *deep learning*.⁵³³

En la práctica, suele resultar beneficioso comenzar el proceso de optimización con algunas iteraciones del gradiente descendente estocástico antes de comenzar el uso de gradientes conjugados. Si empleamos aprendizaje por minibatches, hemos de tener en cuenta que los métodos de optimización de primer orden admiten minibatches más pequeños (p.ej. de 100 ejemplos), mientras que los métodos de segundo orden requieren minibatches mucho más grandes (p.ej. de 10000 ejemplos) para minimizar las fluctuaciones que se producen inevitablemente en las estimaciones de la matriz hessiana (método de Newton), sus aproximaciones (métodos quasi-Newton) o las direcciones conjugadas (gradientes conjugados).

Aprendiendo a optimizar de forma automática

A lo largo de este capítulo hemos visto que, en aprendizaje automático, los problemas de optimización se suelen resolver utilizando alguna forma de gradiente descendente, que da lugar a algoritmos iterativos en los que los parámetros se van actualizando de acuerdo a la siguiente expresión:

$$\Delta x_k = -\eta_k \nabla_x f(x_k)$$

El rendimiento de las muchas variantes que se han propuesto del gradiente descendente se ve afectado por la forma en la que se vaya ajustando la tasa de aprendizaje η_k . Los algoritmos originales se limitaban a establecer un valor fijo para esa tasa de aprendizaje. Propuestas posteriores diseñaban diversas técnicas de ajuste de esa tasa, que podían definirse de forma independiente para cada variable del problema de optimización (tasas de aprendizaje locales). Otros métodos, algo más sofisticados, recurren a información obtenida de las segundas derivadas

⁵²⁹ P. Patrick van der Smagt. Minimisation methods for training feedforward neural networks. *Neural Networks*, 7(1):1 – 11, 1994. ISSN 0893-6080. DOI: 10.1016/0893-6080(94)90052-3

⁵³⁰ Amit Bhaya y Eugenius Kaszkurewicz. Steepest descent with momentum for quadratic functions is a version of the conjugate gradient method. *Neural Networks*, 17(1):65 – 71, 2004. ISSN 0893-6080. DOI: 10.1016/S0893-6080(03)00170-9

⁵³¹ E.M. Johansson, F.U. Dowla, y D.M. Goodman. Backpropagation learning for multilayer feed-forward neural networks using the conjugate gradient method. *International Journal of Neural Systems*, 02(04):291–301, 1991. DOI: 10.1142/S0129065791000261

⁵³² Martin Fodslette Moller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6(4):525 – 533, 1993. ISSN 0893-6080. DOI: 10.1016/S0893-6080(05)80056-5

⁵³³ Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, y Andrew Y. Ng. On Optimization Methods for Deep Learning. En *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 265–272, USA, 2011. Omnipress. ISBN 978-1-4503-0619-5. URL http://www.icml-2011.org/papers/210_icmlpaper.pdf

de la función objetivo, desde el método de Newton, que calcula de forma explícita la matriz hessiana, hasta otras variantes más adecuadas para problemas de optimización a gran escala, como el algoritmo L-BFGS. Sin olvidarnos de los gradientes conjugados que acabamos de ver. En definitiva, el diseño de técnicas de optimización se centra en la creación de reglas de actualización de los parámetros para los algoritmos iterativos con los que resolvemos clases particulares de problemas de optimización.

Nada es gratis en la vida. De la misma que el teorema de Wolpert nos indica que no existe un modelo que sea superior a otros en aprendizaje automático en todas las situaciones posibles, existe un resultado similar para los problemas de optimización.⁵³⁴ En el contexto de los problemas de optimización, ningún algoritmo, en media, ofrecerá resultados mejores que una estrategia completamente aleatoria. Obviamente, eso no quiere decir que no podamos diseñar técnicas específicas que funcionen excepcionalmente bien para el tipo particular de problemas de optimización a los que nos enfrentemos. De hecho, el teorema de Wolpert para la optimización nos indica que esa es precisamente la única estrategia válida para mejorar el rendimiento de las técnicas de optimización.

En otras palabras, siempre podremos diseñar técnicas de optimización que se adapten como un guante al tipo de problemas que tengamos entre manos. Tradicionalmente, ese diseño ha consistido en elaborar reglas de actualización ajustadas manualmente mediante heurísticas más o menos acertadas. Sin embargo, nada nos obliga a tener que diseñar manualmente esas heurísticas. Podemos ver el problema del diseño de técnicas de optimización como un problema más de aprendizaje automático. En este caso, nuestro objetivo es aprender de forma automática una función g , a la que denominaremos optimizador, que se ajuste con precisión al problema de optimización de la función f , la función optimizada:

$$\Delta x_k = -g_k(\nabla_x f(x_k), \phi)$$

donde ϕ es el conjunto de parámetros del optimizador g .

El equipo de Nando de Freitas en Google DeepMind y la Universidad de Oxford ha utilizado esta estrategia para diseñar optimizadores a medida para problemas concretos. El resultado, un algoritmo que aprende a aprender usando el gradiente descendente... con el gradiente descendente.⁵³⁵ Una estrategia que podríamos denominar “meta gradiente descendente” al tratarse de un problema de meta-aprendizaje (la calificación de gradiente descendente de segundo orden podría resultar algo engañosa, al poder confundirse con las técnicas de optimización de segundo orden).

En sus experimentos, para modelar el optimizador g , los investigadores utilizaron, cómo no, una red neuronal. En particular, una red neuronal recurrente capaz de mantener su propio estado y actualizar dinámicamente la función g conforme avanza el entrenamiento de la red: el optimizador g

⁵³⁴ David H. Wolpert y William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, Apr 1997. ISSN 1089-778X. doi: 10.1109/4235.585893

⁵³⁵ Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, y Nando de Freitas. Learning to learn by gradient descent by gradient descent. *arXiv e-prints*, arXiv:1606.04474, 2016. URL <http://arxiv.org/abs/1606.04474>

determina en cada iteración cómo se actualizan los parámetros del modelo que está siendo optimizado, mientras que la señal de error del modelo optimizado también sirve para ajustar los parámetros del optimizador.

En realidad, la idea de aprender reglas para entrenar redes neuronales no es nueva,⁵³⁶ incluso con redes neuronales,⁵³⁷ aunque entonces se aprendía a aprender sin el gradiente descendente usando el gradiente descendente. Otras propuestas de la época intentaban diseñar redes neuronales que fuesen capaces de ajustar sus propios pesos, con éxito limitado.

Utilizando una red neuronal, entrenada usando el gradiente descendente, para que se use como mecanismo de control del aprendizaje de una red neuronal, se obtienen resultados interesantes.

La red neuronal que resuelve el problema que nos interesa la entramos de la forma habitual, de acuerdo al gradiente $\nabla_x f(x)$ y a la regla g aprendida por el optimizador.

La red que aprende el optimizador g controla el aprendizaje de la red que minimiza f y se entrena usando el gradiente de una función de pérdida con respecto a sus parámetros $\nabla_\phi L(\phi)$. En lugar de que esa función de pérdida dependa sólo del resultado final obtenido, $f(x_{final})$, se diseña de forma que se tenga en cuenta la trayectoria de optimización que recorre al optimizar f : $L(\phi) = E[\sum w_k f(x_k)]$, motivo por el que se emplea una red recurrente como optimizador. El uso de una red recurrente, con memoria, le permite al optimizador aprender una regla de actualización dinámica que integra información del historial de gradientes, de forma similar a los momentos.

Al emplear esta estrategia sobre conjuntos de datos estándar, como MNIST, el optimizador creado automáticamente es capaz de mejorar los resultados obtenidos por optimizadores diseñados manualmente. El optimizador neuronal supera al gradiente descendente estocástico convencional, al uso de momentos de Nesterov, al método Adam y al algoritmo RMSprop, todos ellos muy usados en *deep learning*. En ocasiones, el optimizador neuronal aprende a realizar ajustes en sentido contrario a los que realizaría un método como Adam. Aunque sea en un sentido limitado (y difícil de interpretar para nosotros), el ordenador aprende a aprender.

Paralelización de las técnicas de optimización

Cuando, en *big data*, nos enfrentamos a grandes volúmenes de datos, se hace imprescindible la paralelización de las técnicas que utilicemos para analizar los datos de los que disponemos. En la mayor parte de las ocasiones, la paralelización es sencilla y consiste, simplemente, en vectorizar las operaciones que tengamos que realizar de acuerdo al algoritmo secuencial que hayamos decidido utilizar.

Algunos problemas, afortunadamente, pueden descomponerse de forma

⁵³⁶ Samy Bengio, Yoshua Bengio, y Jocelyn Cloutier. On the search for new learning rules for ANNs. *Neural Processing Letters*, 2(4):26–30, 1995. ISSN 1573-773X. doi: 10.1007/BF02279935

⁵³⁷ Thomas Philip Runarsson y Magnus Thor Jonsson. Evolution and design of distributed learning rules. En *2000 IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks*, pages 59–63, 2000. doi: 10.1109/ECNN.2000.886220

sencilla en subproblemas que se pueden resolver por separado. Cuando esa descomposición se consigue con poco (o incluso ningún) esfuerzo, se suele hablar de algoritmos “embarazosamente paralelos” [*embarrassingly parallel*], un término atribuido al cofundador de MATLAB Cleve Moler.⁵³⁸ Reciben este calificativo cuando los problemas se pueden trasladar a un entorno paralelo sin usar técnicas de programación paralela, ya que no existen dependencias entre las tareas que se ejecutan en paralelo y, por tanto, no hacen falta mecanismos de comunicación y coordinación que controlen la ejecución del algoritmo paralelo.

Las operaciones aritméticas con matrices son un ejemplo típico del tipo de problemas que admite el uso de algoritmos embarazosamente paralelos. En los años 80, bibliotecas de álgebra lineal como BLAS [*Basic Linear Algebra Subprograms*] o LINPACK se podían utilizar para ejecutar algoritmos de tipo numérico en los supercomputadores de la época, como ordenadores vectoriales SIMD o multiprocesadores MIMD. Actualmente, la arquitectura de una GPU [*Graphical Processing Unit*] resulta especialmente indicada para facilitar la ejecución paralela de algoritmos que trabajen sobre matrices, utilizando estándares de facto como CUDA, un API creado por NVidia para sus GPU.

Como los algoritmos utilizados en el entrenamiento y uso de redes neuronales están basados en el uso de matrices, resulta casi trivial paralelizar la ejecución de esos cálculos matriciales cuando disponemos de una GPU, aunque la tarea se puede llegar a complicar significativamente cuando se ha de coordinar la ejecución de esos cálculos entre varias GPUs, ya estén todas instaladas sobre una misma máquina usando puentes SLI [*Scalable Link Interface*] o se usen para equipar los distintos nodos de un clúster. Como cada GPU dispone de su propia memoria, hay que diseñar una estrategia adecuada para la comunicación de resultados intermedios de un nodo a otro y coordinar la ejecución de las tareas asignadas a cada nodo.

El extremo opuesto a los problemas “embarazosamente” paralelos son los problemas inherentemente secuenciales, que constan de una secuencia definida de pasos que hay que ejecutar en orden debido a las dependencias existentes entre ellos. Esto es, los problemas inherentemente secuenciales no se pueden paralelizar en absoluto, con lo que no se pueden lograr mejoras de rendimiento invirtiendo en hardware paralelo.

Las técnicas de optimización basadas en el gradiente descendente tienen un carácter evidentemente secuencial. La parallelización de cada iteración del algoritmo es relativamente sencilla, ya que involucra operaciones con matrices, algo que hace idóneas a las GPU para su uso en *deep learning*. Sin embargo, el algoritmo completo de optimización es inherentemente secuencial, pues se trata de un algoritmo de tipo iterativo. Paso a paso, en cada iteración, se va progresando hacia el mínimo.

Cuando se aplican sobre conjuntos de datos enormes, las técnicas

⁵³⁸ Cleve Moler y Michael T. Heath, editores. *Matrix Computation on Distributed Memory Multiprocessors*. Society for Industrial and Applied Mathematics, 1986. ISBN 0898712092

La mayor parte de las técnicas utilizadas en *big data* estarían adscritas a la categoría de “embarazosamente” paralelos. Herramientas de tipo MapReduce, como Hadoop o Spark, paralelizan la ejecución de algoritmos sobre grandes conjuntos de datos descomponiendo esos conjuntos de datos en fragmentos que se procesan localmente por separado para obtener resultados parciales en cada nodo de un clúster [*map*], resultados que luego se combinan en un proceso que involucra comunicación entre los nodos [*reduce*].

iterativas de optimización pueden resultar demasiado lentas cuando se ejecutan secuencialmente. La ejecución síncrona (secuencial) de las etapas de un algoritmo de optimización puede garantizar la convergencia del algoritmo, pero no es idónea en la práctica, por lo que se han propuesto variantes asíncronas. Ejecutar de forma asíncrona (en paralelo) un algoritmo iterativo de optimización puede ayudarnos a conseguir resultados más rápidamente, aunque la convergencia del algoritmo sea peor si la comunicación entre los nodos involucrados en la ejecución paralela del algoritmo no es la óptima.

Veamos algunas de las técnicas que se han propuesto para realizar implementaciones paralelas y distribuidas de algoritmos de optimización como el gradiente descendente estocástico:

Implementación asíncrona paralela del gradiente descendente

Cuando se utiliza un sistema multiprocesador con memoria compartida, SMP [*Shared-Memory Multiprocessor* originalmente, *Symmetric multiprocessing* en la actualidad], la paralelización se puede conseguir mediante paralelismo de datos (cada procesador trabaja sobre un subconjunto diferente de los datos) o paralelismo de parámetros (cada procesador se encarga de una parte del modelo). Aunque la latencia es muy baja en un sistema SMP, ya que la comunicación se realiza a través de la memoria principal del ordenador, compartida entre varios procesadores, el uso de mecanismos de sincronización para garantizar que varios procesadores no escriben a la vez en la misma zona de memoria puede ralentizar la ejecución del algoritmo (un procesador puede pasarse la mayor parte del tiempo esperando a que otro libere un cerrojo para poder acceder a memoria). Por este motivo, puede resultar beneficiosa una implementación asíncrona del algoritmo, en la que cada procesador pueda escribir cuando quiera en memoria, aun con el riesgo de que un procesador se cargue el trabajo de otro sobrescribiendo parte de sus resultados. Bengio y sus colaboradores observaron que estas sobrescrituras, aunque no del todo deseables, sólo introducen algo de ruido en el proceso de optimización y, aparentemente, no ralentizan el entrenamiento de la red.⁵³⁹

Hogwild!,⁵⁴⁰ que en castellano podríamos traducir por “fuera de control”, emplea esta estrategia para implementar métodos de optimización basados en el gradiente descendente estocástico en entornos paralelos... sin utilizar ningún tipo de mecanismo de sincronización. Cada procesador, por separado, muestrea un minilote y ejecuta una iteración del algoritmo SGD, tras lo que escribe sus resultados sin preocuparse de que otros procesadores puedan estar accediendo al mismo tiempo. Esto viola cualquier principio que haya podido aprender de programación concurrente pero, sin embargo, funciona bien en el caso del gradiente descendente estocástico. Cuando las actualizaciones sólo modifican significativamente

⁵³⁹ Yoshua Bengio, Réjean Ducharme, Pascal Vincent, y Christian Janvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, March 2003. ISSN 1532-4435. URL <http://www.jmlr.org/papers/v3/bengio03a.html>

⁵⁴⁰ Benjamin Recht, Christopher Re, Stephen Wright, y Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. En J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *NIPS’2011 Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011. URL <https://goo.gl/qYqnQ3>

una fracción de los parámetros de la red (esto es, los gradientes son dispersos), la implementación asíncrona consigue una tasa de convergencia casi óptima, ya que es poco probable que un procesador sobrescriba el trabajo útil realizado por otros procesadores. Aunque la mejora media obtenida con cada paso del gradiente descendente se vea ligeramente reducida, debido a las sobrescrituras, la tasa mayor de actualizaciones contribuye a que el proceso de entrenamiento progrese más rápidamente.

Implementación asíncrona distribuida del gradiente descendente

Downpour SGD,⁵⁴¹ el “chaparrón” o “aguacero”, es la implementación paralela del gradiente descendente usada por Google en su *framework* DistBelief, predecesor de TensorFlow. En lugar de utilizar la memoria compartida para almacenar los parámetros del modelo, se utiliza un algoritmo distribuido en el que un servidor de parámetros se encarga de gestionar los parámetros del modelo.

En un entorno distribuido, cada nodo mantiene su propia réplica del modelo, que va actualizando usando aprendizaje por minilotes sobre un subconjunto del conjunto de datos. Esto es, los minilotes se muestran, localmente, del fragmento del conjunto de datos asignado a cada nodo.

De forma asíncrona, periódicamente, cada nodo envía al servidor de parámetros los gradientes acumulados localmente. El servidor de parámetros incorpora esos gradientes a los valores de los parámetros del modelo que sirve de referencia (usando la regla de actualización típica del gradiente descendente) y esos valores de referencia se envían, también de forma asíncrona, a los distintos nodos que colaboran en el entrenamiento de la red. Controlando la frecuencia con la que se transmiten gradientes (*push*, de los nodos al servidor de parámetros) y los parámetros actualizados (*fetch*, del servidor de parámetros a los nodos) se puede reducir el sobrecoste [*overhead*] asociado a la implementación paralela del gradiente descendente.

Para evitar el cuello de botella que crea el uso de un servidor centralizado único, se pueden utilizar varios nodos como servidor de parámetros, cada uno de los cuales se encargará, en paralelo, de gestionar una fracción de los parámetros del modelo [*sharding*]. Por tanto, *Downpour SGD* es asíncrono en dos sentidos: cada réplica local del modelo cambia de forma independiente a las demás y cada servidor de parámetros se ejecuta independientemente de los otros que gestionan los otros fragmentos [*shards*] de los parámetros del modelo.

Como las réplicas no se comunican entre sí, sólo a través del servidor de parámetros, los valores de los parámetros que cada una de ellas utiliza están continuamente en riesgo de divergir, lo que puede obstaculizar la convergencia del algoritmo. Cada réplica trabaja calculando sus gradientes

⁵⁴¹ Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, y Andrew Y. Ng. Large scale distributed deep networks. En *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pages 1223–1231. Curran Associates Inc., 2012. URL <https://goo.gl/tnZfdv>

sobre un conjunto de parámetros que está siempre ligeramente anticuado. A cambio, la implementación es robusta frente a fallos en las máquinas. En una implementación síncrona, el fallo de una máquina podría detener todo el proceso de optimización. En esta implementación asíncrona, si falla una réplica, las demás continuarán funcionando sin interrupción.

A diferencia de lo que sucede en una implementación basada en el uso de memoria compartida, como en *Hogwild!*, no existe base teórica alguna que garantice la seguridad del algoritmo al ejecutarse en un entorno completamente distribuido. Además de que cada réplica es ligeramente diferente, la paralelización del servidor de parámetros hace que tampoco tengamos garantías de que todos los parámetros del modelo hayan estado sujetas al mismo número de actualizaciones ni de que esas actualizaciones se hayan realizado en el mismo orden. Aun así, funciona notablemente bien en la práctica.

TensorFlow es el *framework* de Google para la implementación y despliegue de modelos de aprendizaje automático a gran escala, muy popular en *deep learning*, que ha reemplazado a DistBelief, sobre el que se diseñó *Downpour SGD*. En TensorFlow, todos los algoritmos se representan en forma de grafo [*computation graph*] en el que cada nodo representa una operación que recibe tensores como entradas y produce tensores como salida.

Si deseamos ejecutar un algoritmo en un entorno multidispositivo, TensorFlow debe resolver dos problemas relacionados: en qué dispositivo ejecutar cada nodo del grafo y cómo gestionar la comunicación entre los nodos.

■ Asignación de nodos

El algoritmo de asignación utiliza un modelo de costes con estimaciones del tamaño, en bytes, de la entrada y de la salida de cada nodo del grafo, así como del tiempo necesario para su ejecución dado un tensor de entrada. Este modelo de coste se puede estimar de forma estática, usando heurísticas para distintos tipos de operaciones, o medirse empíricamente a partir de ejecuciones anteriores del grafo.

El algoritmo de asignación simula una ejecución del grafo y emplea heurísticas *greedy* para determinar en qué dispositivo debe ejecutarse cada nodo del grafo. Comienza por los nodos iniciales del grafo y, conforme lo va explorando, analiza localmente los efectos de asignar un nodo a cada uno de los dispositivos en los que puede ejecutarse, estimando su tiempo de ejecución en función de las características del dispositivo y los costes de comunicación en los que se incurrirían.

Como resultado, se obtiene el tiempo de ejecución del grafo para cada posible asignación del nodo a un dispositivo y se selecciona la asignación que lo minimiza.

El usuario de TensorFlow también puede indicar explícitamente dónde

Un tensor no es más que *array* multidimensional: un valor escalar (orden 0), el vector que representa una señal de voz (orden 1), la matriz que representa una imagen en blanco y negro (orden 2), el *array* tridimensional que representa una imagen en color con varios canales (orden 3), un el *array* de 4 dimensiones que contiene el conjunto de imágenes en color de un minilote (orden 4).

quiere que se ejecuten los nodos de su grafo computacional, imponiendo restricciones sobre el tipo de dispositivo que desea utilizar (como una GPU), el subconjunto de dispositivos que se pueden considerar (p.ej. los de una subred concreta) o con qué nodos debe estar colocalizado, de forma que podemos indicar cuándo un mismo dispositivo debe encargarse de múltiples tareas.

■ Comunicación entre nodos

Una vez que los todos los nodos del grafo se han asignado a dispositivos concretos, el grafo se divide en subgrafos, uno por dispositivo. Las aristas del grafo que cruzan de x en un dispositivo a y en otro se sustituyen por dos aristas: de x a un nodo emisor [*send*] en el subgrafo del primer dispositivo y de un nodo receptor [*receive*] a y en el subgrafo del segundo dispositivo. La implementación de los nodos emisores y receptores se encarga de coordinar la transferencia de datos de un dispositivo a otro, encapsulando toda la comunicación y simplificando la implementación del resto del sistema. Además, esto permite que la planificación de la ejecución de los distintos nodos del grafo se pueda realizar de forma distribuida: cada dispositivo se encargará única y exclusivamente de ejecutar su subgrafo, sin preocuparse de la sincronización con el resto del grafo, encapsulada en las parejas de nodos emisor/receptor.

DistBelief y TensorFlow no fueron los primeros sistemas de su categoría. Antes de que se hablase de *deep learning*, ya se habían analizado distintas formas de entrenar en paralelo una red neuronal. Algunas de ellas, como Multi-Spert, recurrían a hardware especializado.⁵⁴²

Siguiendo la filosofía de Multi-Spert y DistBelief, el sistema Adam de Microsoft Research también utiliza un servidor de parámetros global.⁵⁴³ Como *Downpour SGD*, Adam explota el paralelismo ofrecido por un clúster de ordenadores intentando minimizar el tráfico de datos en la red, con comunicaciones que se realizan siempre de forma asíncrona. Además, Adam explota el paralelismo que ofrece un sistema SMP, como microprocesadores multinúcleo o multiprocesadores, a nivel de hebras que se comunican de forma asíncrona sin cerrojos, igual que *Hogwild!*.

Muchos grupos utilizan implementaciones asíncronas del gradiente descendente. Algunos llegan a construir supercomputadores especializados con una capacidad de cálculo del orden de petaFLOPS (10^{15} operaciones en coma flotante por segundo). Es el caso de los investigadores de Baidu que construyeron Minwa, que alcanzaron sus minutos de fama por otros motivos.⁵⁴⁴ Minwa era un supercomputador de sólo 36 nodos (frente a las decenas de miles de CPUs del clúster que Google usaba en DistBelief). Eso sí, cada uno de los 36 nodos estaba dotado de un microprocesador Xeon multinúcleo y de 4 GPUs NVidia, además de estar conectado a los demás por medio de una red de fibra óptica InfiniBand con soporte

⁵⁴² Philipp Farber y Krste Asanovic. Parallel neural network training on Multi-Spert. En *Proceedings of 3rd International Conference on Algorithms and Architectures for Parallel Processing*, pages 659–666, 1997. DOI: 10.1109/ICAPP.1997.651531

⁵⁴³ Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, y Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. En *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 571–582, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL <https://goo.gl/Tiys5K>

⁵⁴⁴ Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, y Gang Sun. Deep image: Scaling up image recognition. *arXiv e-prints*, arXiv:1501.02876, 2015. URL <http://arxiv.org/abs/1501.02876>. Withdrawn

para RDMA [*Remote direct memory access*]. En empresas más pequeñas y en entornos académicos, generalmente, no se dispone de los recursos económicos necesarios para construir sistemas de esa escala, pero se pueden construir sistemas similares a partir de componentes COTS [*commercial off-the-shelf*] relativamente más económicos.⁵⁴⁵

No todas las implementaciones distribuidas del gradiente descendente se limitan, digamos, a repartir el trabajo y esperar a que el trabajo realizado en paralelo compense las pérdidas ocasionadas por la falta de coordinación entre los distintos nodos del sistema:

- *En los nodos locales*

Brendan McMahan, de Google, y Matthew Streeter, de Duolingo, propusieron extender los métodos que ajustan automáticamente las tasas de aprendizaje, como AdaGrad, para que tengan en cuenta, no sólo el historial de gradientes anteriores, sino también los retardos que inevitablemente se producen en la propagación de las actualizaciones de los parámetros un sistema distribuido.

Los algoritmos tolerantes a retardos [*delay-tolerant*], como *Adaptive-Revision*,⁵⁴⁶ mejoran la precisión del gradiente descendente asíncrono cuando los retardos en las actualizaciones de los parámetros son grandes en comparación con el tiempo empleado en cada actualización local. Se basan en mantener una suma de los gradientes evaluados desde la última actualización recibida desde el servidor de parámetros hasta la lectura de una nueva actualización, lo que les permite realizar una revisión de los pasos que se dan usando valores antiguos de los parámetros: un pequeño ajuste adicional además del $-\eta \nabla f(x)$ típico del gradiente descendente. Este ajuste les permite corregir las desviaciones ocasionadas por la ejecución local asíncrona de iteraciones del gradiente descendente, que pueden ser significativas y degradan el rendimiento del algoritmo distribuido conforme mayores sean los retardos.

- *En el servidor de parámetros*

Sixin Zhang, Anna Choromanska y Yann LeCun, de la Universidad de Nueva York, propusieron una técnica denominada promediado elástico [*elastic averaging*].⁵⁴⁷ La técnica se basa en permitir que los valores locales con los que trabaja cada nodo fluctúen con respecto a los valores “centrales” almacenados en el servidor de parámetros, que se utilizan de referencia. Las fluctuaciones locales permiten que cada nodo local explore zonas distintas del espacio de parámetros, lo que dota al sistema de capacidad para encontrar mejores óptimos locales.

El gradiente descendente estocástico con promediado elástico, o EASGD [*Elastic Averaging SGD*], acelera el entrenamiento de redes neuronales en comparación con *Downpour SGD*. ¿Cómo? Actualizando los valores centrales (los del servidor de parámetros) mediante una

⁵⁴⁵ Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, y Bryan Catanzaro. Deep Learning with COTS HPC Systems. En Sanjoy Dasgupta y David McAllester, editores, *Proceedings of the 30th International Conference on International Conference on Machine Learning*, volume 28 of *ICML’13*, pages III.1337–III.1345, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/coates13.html>

⁵⁴⁶ Brendan McMahan y Matthew Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger, editores, *NIPS’2014 Advances in Neural Information Processing Systems 27*, pages 2915–2923. Curran Associates, Inc., 2014. URL <https://goo.gl/yQSu8D>

⁵⁴⁷ Sixin Zhang, Anna E Choromanska, y Yann LeCun. Deep learning with Elastic Averaging SGD. En C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, y R. Garnett, editores, *NIPS’2015 Advances in Neural Information Processing Systems 28*, pages 685–693. Curran Associates, Inc., 2015. URL <https://goo.gl/z1REBz>

media móvil, en vez de utilizar, tal cual, los valores recibidos de los diferentes nodos. Estableciendo una analogía física, esa media móvil se puede interpretar como una fuerza elástica que conecta los valores locales de los nodos con sus valores centrales.

Paralelización de otras técnicas de optimización

Las técnicas de optimización basadas en el gradiente descendente no son las únicas para las que se han propuesto algoritmos paralelos, obviamente. Desde hace décadas existen implementaciones paralelas de métodos quasi-Newton como BFGS o L-BFGS.^{548,549} Estas implementaciones se basaban en la utilización de mecanismos de paso de mensajes: originalmente, para transputers (microprocesadores de los años 80 diseñados para construir sistemas paralelos que se programaban con un lenguaje llamado Occam); posteriormente, mediante PVM [*Parallel Virtual Machine*]; y, más recientemente, con el estándar MPI [*Message Passing Interface*].

Más recientemente, en DistBelief, la plataforma de Google que precedió a TensorFlow, también se realizaron implementaciones distribuidas de algoritmos como L-BFGS. Estos algoritmos, diseñados para su uso en aprendizaje por lotes, requieren una ejecución coordinada que no resulta necesaria en algoritmos de aprendizaje por minibatches como la versión asíncrona del gradiente descendente estocástico (*Downpour SGD* en DistBelief). La implementación de Google, denominada *Sandblaster L-BFGS*, de “pulido con chorro de arena”, consiste en que un coordinador único envía mensajes cortos a las réplicas del modelo y al servidor de parámetros para organizar el proceso de optimización por lotes.

El proceso de coordinación no tiene acceso directo a los parámetros del modelo. Se limita a enviar órdenes a los servidores de parámetros que realizan las operaciones sobre los parámetros (productos escalares, multiplicaciones...), que cada servidor de parámetros efectúa de forma independiente sobre su fragmento [*shard*] de parámetros del modelo y cuyos resultados se almacenan localmente. De esta forma, se pueden realizar operaciones sobre conjuntos de parámetros enormes (hasta miles de millones) sin necesidad de incurrir en el coste que supondría tener que transmitirlos a un servidor central.

En una implementación paralela típica de L-BFGS, los datos también se distribuyen entre muchas máquinas, cada una de las cuales es responsable de calcular el gradiente sobre un subconjunto específico del conjunto de datos. Los gradientes se envían al servidor de parámetros, directamente o agregándose parcialmente a través de un árbol (como en MapReduce, Hadoop o Spark). Esta estrategia funciona bien en un clúster homogéneo en el que se reparte la carga de forma equitativa, aunque no así en un clúster heterogéneo, en el que la máquina más lenta

⁵⁴⁸ Sean McLoone y George W. Irwin. Fast parallel off-line training of multilayer perceptrons. *IEEE Transactions on Neural Networks*, 8(3):646–653, May 1997. ISSN 1045-9227. doi: 10.1109/72.572103

⁵⁴⁹ P. K. H. Phua y Daohua Ming. Parallel nonlinear optimization techniques for training neural networks. *IEEE Transactions on Neural Networks*, 14(6):1460–1468, Nov 2003. ISSN 1045-9227. doi: 10.1109/TNN.2003.820670

tendrá esperando al resto. Para balancear mejor la carga, el coordinador asigna a cada una de las n réplicas del modelo una porción pequeña del trabajo, mucho menor que la n -ésima parte del trabajo total. El coordinador va repartiendo trabajo entre los nodos que queden libres y, de esta forma, los nodos más rápidos harán más trabajo que los nodos más lentos.

En ocasiones, para evitar el impacto de tareas cuya ejecución se retrase [*stragglers*], el coordinador recurre a la ejecución especulativa. Le asigna la misma tarea a distintos nodos y se utiliza el resultado de la tarea cuya ejecución termine primero. Es una estrategia habitual en muchos sistemas de *cloud computing* que utilizan MapReduce, donde se les denomina tareas de respaldo [*backup tasks*].⁵⁵⁰

El coordinador, por tanto, actúa como la unidad de control de una CPU, salvo que controla un clúster completo de ordenadores. Se limita a enviar mensajes cortos de control para orquestar el funcionamiento del sistema distribuido. Obviamente, dado el elevado coste de transmitir datos de un nodo a otro, procurará mantener su afinidad a cada nodo asignando a un nodo todas las tareas que involucren la manipulación de un fragmento concreto de los datos, asignará porciones secuenciales de los datos a las mismas tareas y, cuando sea posible, emitirá instrucciones de pre-lectura [*prefetch*] para que los datos estén ya disponibles en el momento que se vayan a utilizar. De esta forma, se enmascaran los retardos asociados a la transmisión de datos en el sistema distribuido y se optimiza la ejecución del algoritmo paralelo.

A diferencia del chaparrón de *Downpour SGD*, que requiere la sincronización frecuente de los parámetros del modelo, lo que consume un elevado ancho de banda en la red, la lluvia fina de órdenes de *Sandblaster* sólo requiere la transmisión de algunos parámetros al comienzo de cada lote (cuando el coordinador los actualiza) y, de vez en cuando, de los gradientes calculados localmente para protegerse de fallos en las réplicas y mejorar la tolerancia a fallos de un sistema que puede tener miles de nodos. El almacenamiento de los parámetros y su manipulación siempre se realiza de forma completamente distribuida.

Comentarios finales

En este capítulo hemos analizado muchas de las técnicas de optimización que se utilizan para entrenar redes neuronales artificiales. Otras muchas variantes de algoritmos numéricos de optimización aquí descritos se pueden encontrar, estudiados con mayor grado de detalle, en algunos libros de texto de cálculo numérico.^{551,552}

Como comentamos anteriormente, el teorema de Wolpert para los problemas de optimización nos indica que, aun no existiendo métodos que sean sistemáticamente mejores que los demás, siempre podemos

⁵⁵⁰ Ganesh Ananthanarayanan y Ishai Menache. *Big Data Analytics Systems*, pages 137–160. Cambridge University Press, 2016. ISBN 1107099005

⁵⁵¹ Jorge Nocedal y Stephen Wright. *Numerical Optimization*. Springer, 2nd edition, 2006. ISBN 0387303030

⁵⁵² David G. Luenberger y Yinyu Ye. *Linear and Nonlinear Programming*. International Series in Operations Research & Management Science. Springer, 4th edition, 2016. ISBN 3319188410

encontrar o diseñar nuestros propios métodos para que se adapten mejor a las características particulares del problema que pretendemos resolver.

En Internet se pueden encontrar descripciones detalladas de muchos métodos numéricos de optimización continua no lineal,⁵⁵³ que son los que se emplean en redes neuronales. Existen también diversos *surveys* o estudios del estado del arte centrados específicamente en técnicas de optimización para *deep learning*. Algunos son más divulgativos⁵⁵⁴ y otros más formales,⁵⁵⁵ aunque siempre resultan informativos.

En situaciones específicas, se han utilizado multitud de técnicas diferentes de optimización para entrenar redes neuronales:

- Técnicas de optimización con restricciones, como las basadas en las condiciones KKT [*Karush–Kuhn–Tucker*], las condiciones necesarias para que la solución a un problema de optimización en programación no lineal sea óptima. El enfoque KKT generaliza el método de los multiplicadores de Lagrange, nombrado en honor del matemático italiano Joseph Louis Lagrange (de nacimiento, se llamaba Giuseppe Luigi). Este método, utilizado por ejemplo por los investigadores de Microsoft Research que trabajan en *deep learning*,⁵⁵⁶ permite resolver problemas de optimización de una función sujeta a una serie de restricciones especificadas mediante inecuaciones (el método de Lagrange admite sólo ecuaciones, igualdades, mientras que el método generalizado permite el uso de desigualdades en las restricciones).
- Otras técnicas de optimización⁵⁵⁷ se basan en la factorización de matrices, más concretamente, en la factorización de matrices no negativas [*NMF: Non-negative Matrix Factorization*]. Aunque estos métodos son más habituales en otras áreas de la Inteligencia Artificial, como en la construcción de sistemas de recomendación, también se han utilizado ocasionalmente en el entrenamiento de redes neuronales.
- En ocasiones, la factorización de matrices se realiza utilizando técnicas como los mínimos cuadrados alternos, ALS [*Alternating least squares*], que se han utilizado tanto para la construcción de sistemas de recomendación de filtrado colaborativo,⁵⁵⁸ que utilizan las preferencias de usuarios similares para ofrecer recomendaciones a un usuario, como en el entrenamiento de redes neuronales en *deep learning*.⁵⁵⁹ El gradiente descendente estocástico es eficiente como método de optimización, pero resulta demasiado sensible tanto a la inicialización de los parámetros como a la forma en la que se van ajustando las tasas de aprendizaje (el tamaño de los pasos que se van dando). En cambio el método de los mínimos cuadrados alternos es más lento, pero más estable.⁵⁶⁰
- Los métodos de descenso por coordenadas [*CD: Coordinate Descent*] pueden ser eficientes, a la vez que mantienen la estabilidad de ALS, que no resulta escalable. En los métodos de optimización de descenso por coordenadas se utiliza la misma estrategia de ALS, fijar el valor

⁵⁵³ Arkadi Nemirovski. Numerical Methods for Nonlinear Continuous Optimization. Technical report, Technion, Israel Institute of Technology, 1999. URL http://www2.isye.gatech.edu/~nemirovs/Lect_OptII.pdf

⁵⁵⁴ Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv e-prints*, arXiv:1609.04747, 2016. URL <http://arxiv.org/abs/1609.04747>

⁵⁵⁵ Léon Bottou, Frank E. Curtis, y Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning. *arXiv e-prints*, arXiv:1606.04838v2, 2017. URL <http://arxiv.org/abs/1606.04838>

⁵⁵⁶ Li Deng y Dong Yu. *Deep Learning: Methods and Applications. Foundations and Trends in Signal Processing*. Now Publishers Inc., 2014. ISBN 1601988141

⁵⁵⁷ Rainer Gemulla, Erik Nijkamp, Peter J. Haas, y Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. En *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, 2011. ISBN 978-1-4503-0813-7. DOI: 10.1145/2020408.2020426

⁵⁵⁸ Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, y Rong Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. En *AAIM'2008 Algorithmic Aspects in Information and Management*, pages 337–348, 2008. ISBN 978-3-540-68880-8. DOI: 10.1007/978-3-540-68880-8_32

⁵⁵⁹ Tetsuya Sakurai, Akira Imakura, Yuto Inoue, y Yasunori Futamura. Alternating optimization method based on nonnegative matrix factorizations for deep neural networks. *arXiv*, arXiv:1605.04639, 2016. URL <http://arxiv.org/abs/1605.04639>

⁵⁶⁰ Prateek Jain, Praneeth Netrapalli, y Sujay Sanghavi. Low-rank matrix completion using alternating minimization. En *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*, STOC'13, pages 665–674, 2013. ISBN 978-1-4503-2029-0. DOI: 10.1145/2488608.2488693

de un conjunto de variables para ajustar los valores de las demás, pero llevado al extremo: se fijan todos los valores menos uno, que es el que se ajusta en cada iteración del algoritmo. De esta forma, se descompone un problema de optimización complejo en subproblemas mucho más sencillos que se resuelven por separado. En vez de ajustar una única variable en cada paso [CD], la estrategia se puede relajar y ajustar un bloque de variables en cada subproblema de optimización [BCD: *Block Coordinate Descent*]. Como sólo hay que ajustar un parámetro, o un pequeño bloque de ellos, el ajuste se puede efectuar, en muchas ocasiones, de forma óptima, especialmente si somos capaces de descomponer un problema de optimización no convexa en una serie de subproblemas de optimización convexa. Y, por supuesto, los algoritmos de este tipo también se pueden parallelizar para resolver problemas a gran escala.^{561,562}

- El método de los multiplicadores de direcciones alternas, ADMM [*Alternating Direction Method of Multipliers*], es un método que permite resolver problemas de optimización convexa descomponiéndolos en subproblemas más pequeños.
- En ocasiones, el problema de entrenar redes neuronales se descompone en subproblemas que consisten en ajustar de forma óptima los parámetros de la red capa a capa. Es la estrategia que utilizan las máquinas de aprendizaje extremo, ELM [*Extreme Learning Machines*], con una sola capa de parámetros ajustables, o el algoritmo OWO-HWO,⁵⁶³ que permite entrenar redes con una sola capa oculta descomponiendo el problema en dos subproblemas: uno de optimización de los pesos de la capa de salida [*OWO: Output Weight Optimization*] y otro de optimización de los pesos de la capa oculta [*HWO: Hidden Weight Optimization*]
- Cuando se conoce información adicional de la estructura del problema de optimización que se pretende resolver, se pueden diseñar métodos a medida que aprovechen esa información.^{564,565}
- Los métodos de continuación utilizan otra estrategia diferente para agilizar el proceso de optimización. En lugar de descomponer el problema en subproblemas que se resuelvan por separado, descomponen el problema original en una serie de problemas de optimización. Definen una serie de funciones objetivo, de forma que las primeras dan lugar a problemas de optimización más sencillos cuyas soluciones se emplean como puntos de partida para problemas de optimización posteriores, más difíciles de resolver si no partísemos de la posición aventajada que nos proporciona la resolución de los problemas previos. Esta estrategia se puede diseñar para que se alcance un mínimo global pese a la presencia de muchos mínimos locales, ya que las funciones de coste más sencillas se construyen “desenfocando” [*blurring*] la función de

⁵⁶¹ Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, y Inderjit Dhillon. Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems. En *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, ICDM'2012, pages 765–774, 2012. ISBN 978-0-7695-4905-7. DOI: 10.1109/ICDM.2012.168

⁵⁶² Peter Richtárik y Martin Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 156(1):433–484, Mar 2016. ISSN 1436-4646. DOI: 10.1007/s10107-015-0901-6

⁵⁶³ Hung-Han Chen, Michael T. Manry, y Hema Chandrasekaran. A neural network training algorithm utilizing multiple sets of linear equations. *Neurocomputing*, 25(1):55 – 72, 1999. ISSN 0925-2312. DOI: 10.1016/S0925-2312(98)00109-X

⁵⁶⁴ Seiya Satoh y Ryohei Nakano. Fast and stable learning utilizing singular regions of multilayer perceptron. *Neural Processing Letters*, 38(2):99–115, October 2013. ISSN 1370-4621. DOI: 10.1007/s11063-013-9283-z

⁵⁶⁵ James Martens y Roger Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. En *Proceedings of the 32nd International Conference on Machine Learning, Volume 37*, ICML'15, pages 2408–2417. JMLR.org, 2015. URL <http://proceedings.mlr.press/v37/martens15.pdf>

coste original. El resultado es un proceso iterativo en el que se van resolviendo problemas cada vez más difíciles, de forma similar a como aprendemos los humanos.^{566,567}

Dada la muy extensa y variada gama de técnicas de optimización que tenemos a nuestra disposición, uno puede no tener claro, llegados hasta aquí, cuál puede ser la técnica más adecuada para enfrentarse al entrenamiento de una red neuronal particular. Aunque no existe una receta sencilla que resulte universalmente válida, sí que podemos utilizar criterios heurísticos para tomar una decisión al respecto.

Para conjuntos de datos pequeños, del orden de un millar de ejemplos de entrenamiento, puede que nos podamos permitir el lujo de utilizar alguna técnica de optimización de orden cuadrático, como BFGS [Broyden–Fletcher–Goldfarb–Shanno], especialmente si disponemos de los recursos computacionales necesarios. El método de Newton, de orden cúbico, resulta demasiado ineficiente para usarlo en *deep learning*, aunque resulte necesario conocerlo para entender de dónde provienen las demás técnicas de optimización de segundo orden.

Cuando dispongamos de un conjunto de datos de tamaño mediano, del orden de decenas de miles de ejemplos, no podremos utilizar algoritmos de optimización que no sean de orden lineal, por lo que recurriremos a variantes de BFGS como L-BFGS [em limited-memory BFGS] o al método de los gradientes conjugados, si optamos por el aprendizaje por lotes. También podemos optar por alguna de las técnicas heurísticas más sofisticadas de ajuste automático de las tasas de aprendizaje, como RMSprop con momentos de Nesterov, de grupo de Hinton, vSGD [*variance-based SGD*] del grupo de LeCun, o AdaSecant, del grupo de Bengio, todas ellas diseñadas para el gradiente descendente estocástico utilizado en aprendizaje con minibatches.

Cuando nuestro conjunto de datos sea realmente grande, con hasta millones de ejemplos, normalmente recurriremos al aprendizaje por minibatches, aprovechando la redundancia que seguramente incluirá nuestro conjunto de datos. Podemos utilizar algunas de las técnicas mencionadas en el párrafo anterior (RMSprop con momentos de Nesterov, vSGD o AdaSecant) o bien decantarnos por técnicas más convencionales de las que se emplean en *deep learning*, entre las que se encuentra Adam o su variante NAdam, que incorpora momentos de Nesterov.

Como sabemos por el teorema de Wolpert, el método más adecuado dependerá de las características de la red neuronal que deseemos entrenar y del tipo de problema al que nos enfrentemos. Puede, incluso, que si optamos por utilizar alguna herramienta de *deep learning*, la elección que realicemos limite bastante nuestras opciones con respecto a qué tipo de optimizador emplear. Bibliotecas populares como TensorFlow, Keras, Theano, Caffe, Torch, MXNet, Deeplearning4j, Microsoft Cognitive Tool-

⁵⁶⁶ Yoshua Bengio, Jérôme Louradour, Ronan Collobert, y Jason Weston. Curriculum learning. En Andrea Pohoreckyj Danyluk, Léon Bottou, y Michael L. Littman, editores, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 41–48. ACM, 2009. ISBN 978-1-60558-516-1. DOI: 10.1145/1553374.1553380

⁵⁶⁷ Faisal Khan, Bilge Mutlu, y Xiaojin Zhu. How do humans teach: On curriculum learning and teaching dimension. En J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *NIPS'2011 Advances in Neural Information Processing Systems 24*, pages 1449–1457. Curran Associates, Inc., 2011a. URL <https://goo.gl/c6TMn6>

kit o el Neural Network Toolbox de MATLAB suelen incluir algunos de los algoritmos más habituales y ser capaces de aprovechar la capacidad de cálculo de una GPU cuando disponemos de ella, aunque la gama de optimizadores que incluyen suele ser más bien reducida.

En ocasiones, no obstante, para mejorar los resultados que se obtienen, la mejor alternativa no siempre es encontrar un algoritmo de optimización que funcione mejor para nuestro problema particular. Muchas de las mejoras que se consiguen en la práctica provienen de modificar el tipo de modelo que estemos entrenando para que resulte más sencillo de optimizar.

Esta observación explica, por ejemplo, la popularidad de las unidades lineales rectificadas, ReLU. Las unidades lineales consiguen que las redes neuronales tengan propiedades que facilitan su entrenamiento. Frente a las unidades sigmoidales que se saturan, en las unidades ReLU el gradiente del error observado puede fluir hacia atrás en la red atravesando muchas más capas.

El uso de saltos en las conexiones de unas capas con otras [*skip connections*] reduce la longitud del camino existente entre una capa de la red y su salida, lo que contribuye a mitigar el problema de la desaparición del gradiente [*vanishing gradient*].⁵⁶⁸

Otra alternativa a nuestra disposición es añadir copias extra de la salida de la red conectadas a capas intermedias ocultas. Esas cabezas extra nos permiten garantizar que las primeras capas ocultas de la red reciben una señal del gradiente lo suficientemente fuerte como para facilitar su entrenamiento.⁵⁶⁹ Aunque, una vez entrenada la red, esas cabezas se desechen, durante el entrenamiento proporcionan pistas adicionales para ajustar los parámetros de las capas ocultas más alejadas de la salida de la red. Una estrategia con un objetivo similar se basa en la introducción de funciones objetivo acompañantes [*companion objective functions*] en las capas ocultas de la red,⁵⁷⁰ que pueden interpretarse como restricciones adicionales que se imponen en el entrenamiento de las capas ocultas, lo que favorece que esas capas ocultas terminen aprendiendo características que puedan ser de ayuda en la resolución del problema para el que estamos entrenando la red neuronal.

⁵⁶⁸ Rupesh Kumar Srivastava, Klaus Greff, y Jürgen Schmidhuber. Highway networks. *ICML'2015 Deep Learning Workshop*, arXiv:1505.00387, 2015b. URL <http://arxiv.org/abs/1505.00387>

⁵⁶⁹ Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, y Andrew Rabinovich. Going deeper with convolutions. En *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9, 2015a. ISBN 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298594

⁵⁷⁰ Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, y Zhuowen Tu. Deeply-Supervised Nets. En Guy Lebanon y S. V. N. Vishwanathan, editores, *AISTATS'2015 Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pages 562–570, San Diego, California, USA, 09–12 May 2015. PMLR. URL <http://proceedings.mlr.press/v38/lee15a.html>

PARTE III. ARQUITECTURAS ESPECIALIZADAS

En la que se analizan algunos de los tipos de bloques que podemos utilizar en la construcción de redes neuronales artificiales, como si de bloques de LEGO se tratase. En particular, se analizan los más habituales en la práctica: los módulos necesarios para resolver problemas de clasificación (softmax) y trabajar con señales tales como sonidos e imágenes (redes convolutivas).

Softmax

Los problemas de aprendizaje supervisado más habituales son los problemas de clasificación, cuando se intenta predecir el valor de un atributo discreto, y los problemas de regresión, cuando se intenta predecir el valor de un atributo continuo. Las redes neuronales multicapa se pueden emplear directamente para resolver ambos tipos de problemas si diseñamos adecuadamente la capa de salida de la red:

- *En los problemas de regresión, basta con que utilicemos neuronas lineales, de forma que su valor de salida no esté acotado por la función de activación de las neuronas de salida.*
- *En los problemas de clasificación, se podrían utilizar directamente neuronas sigmoidales, si bien veremos que el uso de capas softmax suele funcionar mejor en la práctica si lo que deseamos es modelar la distribución de probabilidad de las clases de un problema.*

Supongamos que deseamos resolver un problema de clasificación utilizando redes neuronales. Cuando nuestro problema de clasificación es binario, nos basta con incluir una única neurona en la capa de salida de la red. Para esa neurona podemos elegir una función de activación sigmoidal como la función logística. La salida y de la neurona la podemos interpretar como la probabilidad de que un ejemplo pertenezca a la clase positiva, siendo $1 - y$ la probabilidad de que pertenezca a la clase negativa. Ahora bien, si nuestro problema de clasificación no es binario, tendremos que ser capaces de diferenciar entre k clases diferentes, con $k > 2$. Una salida escalar, proveniente de una única neurona de salida, no nos servirá. Necesitaremos más de una neurona de salida para estimar las probabilidades asociadas a las distintas clases de nuestro problema.

Supongamos, además, que recurrimos a la configuración más tradicional de red neuronal multicapa, aquélla formada por unidades sigmoidales que entrenamos utilizando el error cuadrático observado en la capa de salida. Esto es, la función de activación de las neuronas de la capa de salida será la función logística:

$$y = f(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Como ya sabemos, la derivada de la función logística se puede expresar elegantemente en función de la propia función logística:

$$f'(z) = \frac{dy}{dz} = \frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

Para entrenar la red, utilizaremos el error cuadrático, que para un problema de clasificación binario calcularemos a partir del nivel de activación de la única neurona de la capa de salida de la red:

$$SE = \frac{1}{2} (t - y)^2$$

donde y representa la salida de la red y t el valor deseado, que viene dado por los datos de nuestro conjunto de entrenamiento.

Recordemos, una vez más, que el entrenamiento de la red se realiza en la dirección del gradiente descendente del error, motivo por el que hemos de calcular la derivada del error (y por el que añadimos artificialmente el término $1/2$ en la expresión anterior):

$$\nabla SE = \frac{\partial SE}{\partial y} = -(t - y)$$

Una vez disponemos del gradiente del error, usamos un algoritmo iterativo para ajustar los parámetros de la red:

$$\Delta w_{ij} = -\eta \frac{\partial SE}{\partial w_{ij}}$$

En el caso de nuestra única neurona de salida, las actualizaciones de los pesos serán de la forma:

$$\begin{aligned} \Delta w_{ij} &= -\eta \frac{\partial SE}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_{ij}} \\ &= -\eta [-(t - y)] f'(z) x_i \\ &= \eta x_i f'(z) (t - y) \end{aligned}$$

En la expresión interviene la derivada de la función de activación, que sabemos que es prácticamente nula en cuanto nos salimos de la zona lineal de la neurona sigmoidal. Como consecuencia, si deseamos que la salida de la red sea 1 cuando su salida actual es cercana a 0 (o viceversa), que es precisamente la situación que corresponde a un error de clasificación, la derivada casi nula de la función de activación hará que las actualizaciones Δw_{ij} de los parámetros de la red sean muy pequeñas, lo que ralentiza notablemente el entrenamiento de la red.

En condiciones normales, cuando el error de la red no sea demasiado grande, el gradiente descendente funcionará adecuadamente. Sin embargo, cuando la red esté completamente equivocada, se necesitarán muchas iteraciones del algoritmo de optimización basado en el gradiente descendente para corregir el error cometido por la red. Aunque el error, eventualmente, se termine corrigiendo, el aprendizaje será muy lento. Esto es algo que no se corresponde a la forma en que normalmente aprendemos nosotros. Cuando cometemos errores muy graves es, precisamente,

cuando más rápido aprendemos (y, además, no se nos olvida nunca). En cambio, cuando nuestros errores no son especialmente llamativos, hasta el punto de que ni siquiera lleguemos a ser conscientes de ellos, es cuando más trabajo nos cuesta modificar nuestros hábitos y corregir nuestros sesgos.

Si entrenamos una red de unidades sigmoidales usando el error cuadrático, no obstante, su comportamiento es justamente el contrario. Los pequeños errores se corrigen con relativa facilidad pero los grandes errores requieren que el error se cometa infinidad de veces. Cada vez que se comete el error, se realiza una mínima corrección, que no impide que la red vuelva a equivocarse en el futuro.

¿De dónde proviene ese comportamiento patológico que se observa en el entrenamiento de una red? De las derivadas parciales de la función de coste con respecto a los parámetros de la red, $\partial E / \partial w_{ij}$, que toman valores muy cercanos a cero en cuanto interviene la derivada de la función de activación sigmoidal, como sucede en el caso del error cuadrático *SE*. Si usamos la función logística como función de activación, su derivada es prácticamente plana cuando el nivel de activación de la neurona se acerca a los niveles que queremos que la red sea capaz de reproducir (0 y 1).

Una forma casera de mitigar parcialmente el problema consiste en utilizar otros niveles de activación diferentes para las neuronas de salida. Por ejemplo, en lugar de establecer $t = 1$ para la clase positiva y $t = 0$ para la clase negativa, podríamos utilizar valores intermedios como $t = 0.9$ y $t = 0.1$. De esta forma, evitaríamos las zonas más planas de la derivada de la función de activación y el entrenamiento de la red progresaría más rápidamente. Sin embargo, ya no sería correcta una interpretación probabilística de la salida de la red.

En realidad, las ideas clave que sirven de base para las redes neuronales artificiales no han cambiado demasiado desde los años 80, cuando se popularizó el uso de *backpropagation* para entrenar redes neuronales multicapa. Las mejoras notables que se han conseguido en cuanto a su rendimiento se deben a dos factores principales: conjuntos de datos más grandes y mayor capacidad de cálculo, lo que permite entrenar redes mucho más grandes. Desde un punto de vista algorítmico, las principales diferencias de las redes neuronales actuales con respecto a las que ya se usaban en los años 80 están en el uso de funciones de activación no sigmoidales, como en las unidades lineales rectificadas, ReLU, y en la introducción de funciones de coste diferentes al error cuadrático.

Aunque el error cuadrático SSE o MSE era popular en los años 80 y 90, fue gradualmente reemplazado por funciones de error basadas en la entropía cruzada y en el principio de máxima verosimilitud, conforme se popularizaban las técnicas probabilísticas entre los investigadores que trabajaban en minería de datos y aprendizaje automático. Como veremos

a continuación, el uso de la entropía cruzada como función de coste nos permitirá mejorar notablemente el entrenamiento de redes neuronales con unidades sigmoidales, evitando los problemas asociados a la saturación de la función de activación que provocaba la ralentización del proceso de aprendizaje de la red al utilizar el error cuadrático.

Más adelante, veremos cómo podemos aprovechar el hecho de que, en un problema de clasificación, queremos asignar probabilidades a clases mutuamente excluyentes. Haciendo que la red genere un conjunto de salidas cuya suma sea siempre 1, para que represente una distribución de probabilidad sobre un conjunto finito de alternativas discretas (las clases de nuestro problema), construiremos clasificadores softmax.

La entropía cruzada como función de coste

Como ya sabemos, las actualizaciones de los pesos de la red se realizan en función de la derivada del error con respecto a esos pesos, $\partial E / \partial w_{ij}$. Aplicando la regla de la cadena, dicha derivada se descompone en tres factores: la derivada del error con respecto a la salida ($\partial E / \partial y$), la de la salida con respecto a la entrada neta ($\partial y / \partial z$) y la de la entrada neta con respecto a los pesos ($\partial z / \partial w_{ij}$). Como vimos, el factor problemático es el asociado a la función de activación de la neurona: $\partial y / \partial z$.

En el caso de una neurona sigmoidal que utilice la función logística, $\partial y / \partial z = \partial \sigma(z) / \partial z = \sigma(z)(1 - \sigma(z))$. Si queremos eliminar ese factor problemático de la fórmula de actualización de los pesos de la red, tendremos que actuar sobre los otros dos factores que intervienen en la actualización de los pesos. El factor $\partial z / \partial w_{ij}$ no es algo sobre el que tengamos libertad, pero sí podemos jugar con la función de error.

Una forma de conseguir anular el impacto de la derivada nula de la función de activación de la neurona es incluir dicho factor como denominador de la derivada de la función de error. Esto es, teniendo en cuenta que $y = \sigma(z)$, definimos una función de error como la siguiente:

$$\frac{\partial CE}{\partial y} = -\frac{t - y}{y(1 - y)}$$

a partir de la función de error a la que ya estábamos acostumbrados:

$$\frac{\partial SE}{\partial y} = -(t - y)$$

Al incorporar esa función de error en nuestra fórmula de actualización de los pesos, desaparece por completo el término que causaba problemas en el entrenamiento de la red:

$$\begin{aligned}\Delta w_{ij} &= -\eta \frac{\partial CE}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_{ij}} \\ &= -\eta [-(t-y)] x_i \\ &= \eta x_i (t-y)\end{aligned}$$

La actualización de los pesos dependerá ahora de la magnitud del error observado y de las entradas recibidas, pero no de la derivada de la función de activación de la neurona. La red será capaz de aprender más rápidamente cuanto mayor sea el error cometido, justo como hacemos nosotros (o deberíamos hacer).

Ahora bien, ¿cuál es esa misteriosa función de error que elimina el problema que observábamos en el entrenamiento de la red? Como conocemos su derivada, sólo tenemos que integrar la siguiente función:

$$f(y) = \frac{\partial CE}{\partial y} = \frac{y-t}{y(1-y)}$$

Con un poco de cálculo, descubrimos cuál es la función de error buscada:

$$\begin{aligned}CE &= \int f(y)dy = \int \frac{y-t}{y(1-y)} dy \\ &= \int \frac{y}{y(1-y)} dy - \int \frac{t}{y(1-y)} dy \\ &= \int \frac{1}{1-y} dy - t \int \frac{1}{y(1-y)} dy \\ &= \int \frac{1}{1-y} dy - t \int \left(\frac{1}{y} + \frac{1}{1-y}\right) dy \\ &= -\log(1-y) - t (\log y - \log(1-y)) \\ &= (t-1) \log(1-y) - t \log(y)\end{aligned}$$

Dicha función de error, que se utiliza para sustituir al error cuadrático *SE* en problemas de clasificación, se denomina entropía cruzada [*CE: Cross-entropy*]. Normalmente, reordenando los términos, se suele expresar de la siguiente forma:

$$CE = -[t \log(y) + (1-t) \log(1-y)]$$

Para entender por qué recibe ese peculiar nombre y cómo podemos justificar su uso en la práctica, tendremos que familiarizarnos con algunos conceptos de Teoría de la Información, una de las ramas formales de la Informática.

Curso rápido de Teoría de la Información

La Teoría de la Información estudia el almacenamiento y la transmisión de información codificada en forma de bits. Sus orígenes se remontan

a los trabajos de Claude Shannon en los Laboratorios Bell. En su famoso artículo de 1948,⁵⁷¹ Shannon identifica los elementos básicos de la comunicación: una fuente de información que produce un mensaje, un transmisor que utiliza el mensaje para crear una señal, un canal a través del cual se puede transmitir esa señal, un receptor que transforma la señal recibida en el mensaje deseado y un destinatario que recibe el mensaje. También desarrolla el concepto de entropía de la información e introduce el término bit como unidad de información (término que atribuye al matemático John Tukey).

Entropía

Imaginemos que lanzamos una moneda al aire. ¿Cuánta información obtenemos al observar el resultado del lanzamiento? Si la moneda no está trucada, esperamos que el resultado sea cara la mitad de las veces y cruz la otra mitad. Por definición, un bit.

Ahora bien, si nuestra moneda está trucada (y lo sabemos de antemano), lo más probable es que el resultado sea, digamos, cara la mayor parte de las veces. Nos sorprendería más que el resultado fuese cruz al lanzar la moneda trucada y la diferencia de probabilidades nos permitiría tratar de embauchar a algún pobre incauto. Dicho de otro modo, la obtención del resultado más improbable (cruz) nos aporta más información, entendiendo por información nuestra sorpresa al descubrir algo.

Teniendo lo anterior en cuenta, podemos tratar de cuantificar la información de un evento que se puede producir con probabilidad p utilizando la siguiente fórmula:

$$I = -\log_2 p$$

Cuando utilizamos logaritmos en base 2, la información se mide en bits [*binary units*], a los que ya estamos acostumbrados por su uso en ordenadores digitales. Si utilizamos logaritmos naturales o neperianos, en base e , la información se mide en nats [*natural units*]. Menos común es el uso de logaritmos en base 10, en cuyo caso estaríamos hablando de dits [*decimal units*], bans (término inventado por Alan Turing y “Jack” Good mientras descifraban el código de las máquinas Enigma en Bletchley Park), o hartleys, en honor de Ralph Hartley, que sugirió en 1928 medir la información utilizando como base logarítmica el número de alternativas diferentes (10 para un dígito decimal).⁵⁷² Igual que los dits se suelen denominar hartleys, en ocasiones, a los bits se les llama shannons. Como sólo estamos cambiando la base del logaritmo, la relación entre las diferentes unidades de información es inmediata:

$$1\text{bit} = 1Sh \approx 0.693nat \approx 0.301Hart$$

Una vez que hemos definido nuestra unidad de información, el bit de ahora en adelante, podemos medir la información media producida

⁵⁷¹ Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1948.tb01338.x

⁵⁷² Ralph V.L. Hartley. Transmission of information. *Bell System Technical Journal*, 7(3):535–563, July 1928. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1928.tb01236.x

por una fuente de datos determinada. Si dicha fuente de datos emite N símbolos diferentes, cada uno de ellos con una probabilidad asociada p_k , su entropía se define como:

$$H = - \sum_{k=1}^N p_k \log_2 p_k$$

Más formalmente, la entropía de una variable aleatoria X , a la que denominamos $H(X)$, es una medida del número esperado de bits necesarios para representar el resultado de un evento $x \sim X$.

- Si el resultado se conoce de antemano (probabilidad 1), no existe aleatoriedad. No hace falta representar nada y el número de bits necesario será 0, por lo que la entropía será mínima: $H(X) = 0$.
- Si el resultado es desconocido, nos gustaría representarlo, en bits, de la forma más eficiente posible. Eso implica utilizar menos bits para los eventos más probables y más bits para los eventos más improbables. La entropía $H(X)$ nos da el número esperado de bits para cualquier $x \sim X$.
- Si el resultado es completamente aleatorio y todos los eventos son equiprobables, la entropía es máxima: $H(X) = -\log_2 1/N = \log_2 N$.

En resumen, la entropía de la información cuantifica la incertidumbre en la predicción del valor de una variable aleatoria. Si, para dos variables aleatorias, Y y Z , observamos que $H(Y) < H(Z)$, podemos concluir que Y es más fácil de predecir que Z .

Entonces, ¿por qué se denomina entropía y no de otra forma más directa, como contenido medio de información? La razón hay que buscarla en el parecido entre la fórmula obtenida por Shannon y una conocida fórmula de mecánica estadística, la entropía de Gibbs. La conexión entre la Termodinámica y la Teoría de la Información proviene de la ecuación formulada por Ludwig Boltzmann:

$$S = k_B \ln W$$

donde S es la entropía termodinámica de un macroestado (definido por parámetros como su temperatura, volumen o energía), k_B es la constante de Boltzmann (que relaciona la energía cinética de las partículas de un gas con su temperatura, $\approx 1.38 \times 10^{-23} J/K$) y W es el número de microestados (combinaciones de partículas en distintos estados de energía), asumiendo que todos los microestados son equiprobables ($p_k = 1/W$). Dada la probabilidad p_k de cada microestado k , la entropía del sistema viene dada por la entropía de Gibbs:

$$S = k_B \sum_k p_k \ln p_k$$

Cuando la probabilidad p_k de un evento es nula, asumimos que su contribución a la entropía es nula. Aunque el logaritmo no esté definido para el valor 0, $\lim_{x \rightarrow 0} x \log(x) = 0$.

La entropía de Gibbs para un sistema clásico (una colección de partículas) con un conjunto discreto de microestados se calcula, pues, con la misma fórmula que halló Shannon en sus estudios sobre la transmisión de información, de ahí que tomase prestado el término entropía.

En términos de Teoría de la Información, la entropía termodinámica de un sistema se puede interpretar como la cantidad de información que resulta necesaria para determinar un microestado, dado un macroestado. En otras palabras, la entropía termodinámica no es más que una manifestación física de la aplicación de la teoría de Shannon: la entropía termodinámica es proporcional (por la constante de Boltzmann) a la cantidad de información necesaria para definir el estado microscópico detallado de un sistema (su microestado) a partir de una descripción del sistema en términos de sus variables macroscópicas (su macroestado). Por ejemplo, el suministro de calor a un sistema aumenta su entropía termodinámica porque se incrementa su número de posibles estados microscópicos compatibles con mediciones macroscópicas, lo que nos obliga a hacer más larga cualquier descripción completa de su estado.

Esta conexión entre la Teoría de la Información y la Termodinámica nos ofrece resultados verdaderamente interesantes. El principio de Landauer, propuesto por el físico Rolf Landauer cuando trabajaba para IBM,⁵⁷³ establece una cota inferior para la cantidad de calor que un ordenador ha de generar para procesar una cantidad dada de información. Esto es, existe un límite teórico para el consumo de energía necesario para realizar cualquier cómputo: “cualquier manipulación de información lógicamente irreversible, como el borrado de un bit..., debe ir acompañada por un incremento correspondiente de la entropía... en el aparato de procesamiento de la información o en su entorno”.

El principio de Landauer es una consecuencia lógica de la segunda ley de la Termodinámica (la entropía de un sistema aislado nunca puede disminuir) cuando se combina con la definición termodinámica de temperatura. Si el número de posibles estados de un cómputo disminuye conforme avanza el cálculo (irreversibilidad lógica), su entropía disminuye. Sin embargo, esta disminución tiene que ir necesariamente acompañado de un aumento del número de posibles estados físicos correspondientes a cada estado lógico para compensar, ya que la entropía global no puede disminuir. Sin embargo, la entropía máxima de un sistema físico acotado es finita (principio holográfico). Para evitar que se alcance ese máximo durante la realización de un cálculo prolongado, el aumento de entropía no puede quedar limitado al dispositivo de cómputo. La entropía tiene que ser eventualmente expulsada al entorno exterior. Para un entorno a temperatura T , se debe emitir al entorno una energía $E = ST$ si se le añade una cantidad de entropía S . Cuando en un cálculo se pierde un bit, la cantidad de entropía generada debe ser, al menos, $k_B \ln 2$. Por tanto, la energía que se debe emitir establece un mínimo para la energía

⁵⁷³ Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, July 1961. ISSN 0018-8646. DOI: 10.1147/rd.53.0183

Cuando no se borra información, en principio, puede lograrse que la computación sea termodinámicamente reversible y no requerir la liberación de calor, lo que ha espoleado es estudio de la denominada computación reversible [*reversible computing*].

necesaria para borrar un bit de información, el límite de Landauer:

$$\Delta E \geq k_B T \ln 2$$

donde k_B es la constante de Boltzmann, T es la temperatura en grados Kelvin y $\ln 2$ es el logaritmo natural de 2 (aproximadamente, 0.693).

El límite de Landauer, a temperatura ambiente (20°C o 293.15K), corresponde a una energía de 0.0172 eV (electronvoltios) o 2.8 zJ (zeptojulios, 10^{-21} julios). Teóricamente, un bit de la memoria de un ordenador podría estar cambiando de estado mil millones de veces por segundo con un consumo de energía de sólo $2.8\text{ billionésimas de vatio}$ (esto es, 2.8 pJ/s). Los ordenadores actuales no son, ni mucho menos, tan eficientes. Requieren millones de veces más energía, por lo que aún queda un amplio margen de mejora en ese aspecto.

Sin embargo, nos acercamos mucho más al límite físico de $0.7k_B T$ julio cuando hablamos del procesamiento de información en sistemas biológicos (nosotros).

Una molécula llamada adenosín trifosfato o ATP es el principal almacén y fuente de energía para la mayoría de las funciones celulares. La hidrólisis natural de una molécula de ATP libera $25k_B T$ julios de energía, cerca del límite termodinámico impuesto por el límite de Landauer. Nos acercamos a este límite cuando una proteína de las que se encuentran en las membranas celulares modifica su conformación para codificar un cambio de estado (un bit). Por ejemplo, el receptor adrenérgico $\beta 2$ es un receptor asociado a la proteína G que se activa por las catecolaminas adrenalina (epinefrina) y noradrenalina (norepinefrina). El receptor $\beta 2$ consume 3 moléculas de ATP en su fosforilación, lo que proporciona una eficiencia de $75k_B T$ julios por bit. La proteína G, por su parte, ejerce de transductor de señales a través de la membrana celular y sólo consume el equivalente a 1 ATP cuando hidroliza su GTP (guanosín trifosfato) a GDP (guanosín difosfato), moléculas de las que proviene su nombre, por lo que su eficiencia es de $25k_B T$ julios por bit. ¿Qué impide que esas moléculas operen más cerca del límite termodinámico? Ese límite termodinámico, de $0.7k_B T$ julios, es el coste mínimo de registrar el cambio de estado de un bit. Sin embargo, no incluye el coste necesario para transmitirlo. Para enviar un bit a través de la membrana celular, el receptor $\beta 2$ mueve una de sus hélices 1.4 nm , mientras que la proteína G tiene que abrir una de sus secciones 110° . Ambos movimientos consumen energía. En cualquier caso, tanto la proteína G como el receptor $\beta 2$ procesan un bit de información por menos de lo que cuesta un simple enlace covalente (del orden de $100k_B T$).

En una neurona, cuando se abre un canal de sodio durante un milisegundo, se admiten unos 6000 iones de sodio (Na^+), que luego han de expulsarse utilizando bombas de sodio-potasio, cuyo funcionamiento requiere la hidrólisis de 2000 moléculas de ATP. La eficiencia de la con-

La fosforilación, que consiste en la adición de grupos fosfato a una molécula, es uno de los principales mecanismos de regulación de la actividad de las proteínas.

versión de la energía química proporcionada por las moléculas de ATP en energía eléctrica suministrada por el canal de iones es bastante alta, del orden del 50 %. No obstante, un canal de iones requiere 2000 veces más energía que una proteína G. Es el precio que hay que pagar para conseguir que una señal se pueda propagar con velocidad más allá del entorno molecular de una membrana celular.

Entropía condicionada e información mutua

Volvamos a la definición de entropía de la información. Para cuantificar la incertidumbre de una probabilidad condicionada $P(Y|X)$, podemos definir una entropía condicionada $H(Y|X)$. La entropía condicionada o “equivocación” $H(Y|X)$ indica la incertidumbre de Y cuando observamos X o, lo que es equivalente, la cantidad de información que necesitamos para describir Y cuando conocemos el valor de X .

Matemáticamente, se puede definir de la siguiente forma, calculando la esperanza sobre los distintos valores de X :

$$\begin{aligned} H(Y|X) &= \sum_x p(x)H(Y|X=x) \\ &= -\sum_x p(x)\sum_y p(y|x)\log_2 p(y|x) \\ &= -\sum_{x,y} p(x)p(y|x)\log_2 p(y|x) \\ &= -\sum_{x,y} p(x,y)\log_2 p(y|x) \\ &= -\sum_{x,y} p(x,y)\log_2 \frac{p(x,y)}{p(x)} \end{aligned}$$

Intuitivamente, podemos ver que $H(Y|X) = 0$ cuando el valor de X determina el valor de Y : cuando conocemos X , no existe incertidumbre en Y y, por tanto, la entropía condicionada es cero. Por otro lado, si las variables X e Y son independientes, la entropía condicionada $H(Y|X)$ es la misma que la entropía original de Y , $H(Y|X) = H(Y)$, ya que conocer X no nos ayuda a reducir la incertidumbre de Y .

Otra medida útil relacionada es la información mutua, definida sobre dos variables aleatorias. Dadas dos variables aleatorias X e Y , la información mutua $I(X, Y)$ es la reducción de la entropía de X debida al conocimiento de Y . Matemáticamente,

$$I(X, Y) = H(X) - H(X|Y)$$

Se trata de una medida simétrica de correlación, por lo que también se puede escribir como

$$I(X, Y) = H(Y) - H(Y|X)$$

Se puede observar que la información mutua $I(X, Y)$ será mayor cuanto más correlacionadas estén X e Y , mientras que será menor cuanto menos relacionadas estén X e Y . Cuando las dos variables son independientes, $H(Y|X) = H(Y)$, por lo que su información mutua es cero (conocer el valor de una no nos ayuda a reducir la entropía de la otra). El máximo de $I(X, Y)$ se alcanza cuando Y viene completamente determinada por X , caso en el que $H(Y|X) = 0$ y, por tanto, $I(X, Y) = H(X)$.

Entropía relativa: Divergencia Kullback-Leibler

La entropía relativa, más conocida como divergencia Kullback-Leibler, mide como difiere una distribución de probabilidad de otra. Es una medida de divergencia dirigida, no simétrica, en la que se mide la diferencia de una distribución de probabilidad p con respecto a una distribución de probabilidad de referencia q . La divergencia de q a p , también conocida como entropía relativa de p con respecto a q , se define como

$$D_{KL}(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)} = -\sum_x p(x) \log \frac{q(x)}{p(x)}$$

Se trata de una medida propuesta en 1951 por los matemáticos y criptoanalistas Solomon Kullback y Richard Leibler en 1951, si bien Kullback prefería originalmente la denominación “información de discriminación”.

La divergencia Kullback-Leibler de q a p es la esperanza de la diferencia logarítmica entre las distribuciones de probabilidad p y q , calculada usando las probabilidades de p . Sólo está definida si $q(x) = 0$ implica $p(x) = 0$ para todos los valores en los que $q(x) = 0$. Como la entropía, su valor nunca es negativo, $D_{KL}(p \parallel q) \geq 0$, resultado conocido como desigualdad de Gibbs. La divergencia KL alcanza su mínimo, $D_{KL}(p \parallel q) = 0$, sólo cuando $p = q$.

La divergencia KL no es simétrica ni satisface la desigualdad triangular. Pese a que no se trata realmente de una métrica de distancia, suele ser útil interpretar la divergencia KL como una especie de “distancia” entre dos distribuciones de probabilidad, una forma de cuantificar las diferencias entre dos distribuciones.

En términos de Teoría de la Información, podemos interpretar la divergencia de q a p , $D_{KL}(p \parallel q)$, como el número extra de bits que debemos transmitir para identificar un valor concreto cuando se utiliza una codificación correspondiente a la distribución de probabilidad q en lugar de una codificación basada en la distribución real p de los valores que deseamos transmitir. En otras palabras, el número extra de bits necesario para codificar muestras de p utilizando un código optimizado para q en lugar de un código optimizado para p .

En aprendizaje automático, la divergencia de q a p , $D_{KL}(p \parallel q)$, en ocasiones se denomina ganancia de información si se usa p en lugar de q o entropía relativa de p con respecto a q . En términos bayesianos,

$D_{KL}(p \parallel q)$ es una medida de la información que ganamos al revisar nuestras creencias desde la distribución de probabilidad a priori, q , hasta la distribución de probabilidad a posteriori, p .

Usualmente, estaremos hablando de las siguientes distribuciones de probabilidad: la distribución de probabilidad real asociada a los datos observados, \hat{p}_{data} , y la distribución de probabilidad asociada al modelo con el que describimos (o aproximamos) los datos, \hat{p}_{model} . La divergencia $D_{KL}(\hat{p}_{data} \parallel \hat{p}_{model})$ mide la cantidad de información que se pierde al utilizar \hat{p}_{model} para aproximar \hat{p}_{data} . Por tanto, nos interesará minimizar la divergencia $D_{KL}(\hat{p}_{data} \parallel \hat{p}_{model})$ para encontrar una distribución \hat{p}_{model} que sea lo más cercana posible a la distribución real \hat{p}_{data} .

La métrica de información de Fisher es otra forma de medir la cantidad de información que una variable aleatoria observable X aporta acerca de los parámetros θ de una distribución que modela X . Matemáticamente, la información de Fisher está directamente relacionada con la divergencia Kullback-Leibler. La información de Fisher puede verse como una forma infinitesimal de la entropía relativa (esto es, de la divergencia KL). En particular, cuando se expresa en forma matricial, la matriz de información de Fisher es la matriz Hessiana que contiene las segundas derivadas de la divergencia KL.

La información mutua, que introdujimos al hablar de la entropía condicionada, también se puede expresar en términos de la entropía relativa. La información mutua entre dos variables aleatorias X e Y es la divergencia Kullback-Leibler del producto de las distribuciones marginales, $p(x)p(y)$, a la distribución de probabilidad conjunta de las dos variables, $p(x, y)$:

$$I(X, Y) = D_{KL}(p(x, y) \parallel p(x)p(y))$$

Es decir, la información mutua es el número extra de bits que han de transmitirse para identificar x e y si se codifican por separado utilizando sus distribuciones marginales de probabilidad en lugar de su distribución conjunta. O, de forma equivalente, conocida la distribución conjunta $p(x, y)$, el número extra de bits que, en media, han de enviarse para identificar el valor de Y cuando el valor de X aún no lo conoce el receptor.

Entropía cruzada

Tras nuestro recorrido por los dominios de la Teoría de la Información, llegamos a la medida que nos permite definir una función de coste adecuada para resolver problemas de clasificación utilizando redes neuronales artificiales: la entropía cruzada.

Al comienzo de este capítulo identificamos un problema que tenían las unidades sigmoidales para aprender a resolver problemas de clasificación. Fuimos capaces de derivar analíticamente una función de coste que

eliminaba dicho problema. En particular, nuestra función de coste era de la forma:

$$CE = -[t \log(y) + (1-t) \log(1-y)]$$

donde t era el valor de salida deseado e y la salida proporcionada por una neurona de tipo sigmoidal. Podemos ver el funcionamiento de dicha neurona como si de un clasificador binario se tratase, de forma que $t = 1$ corresponda a ejemplos de la clase positiva P y $t = 0$ corresponda a ejemplos de la clase negativa N . Para un ejemplo particular, sólo uno de los dos términos de la función de coste será distinto de cero.

Ahora bien, si agregamos los resultados individuales para un conjunto de datos de entrenamiento completo, t pasaría a ser $\hat{p}_{data}(P)$ y y correspondería a $\hat{p}_{model}(P)$. Si tenemos en cuenta que, en un problema de clasificación binaria, $p(N) = 1 - p(P)$, $1 - t$ se transformaría en $\hat{p}_{data}(N)$ y, por último, $1 - y$ sería $\hat{p}_{model}(N)$. La expresión anterior queda como sigue:

$$CE = -[\hat{p}_{data}(P) \log \hat{p}_{model}(P) + \hat{p}_{data}(N) \log \hat{p}_{model}(N)]$$

Esta función de coste, definida para un problema de clasificación binario, se puede generalizar para más de dos clases como

$$CE = -\sum_x \hat{p}_{data}(x) \log \hat{p}_{model}(x)$$

Si, por conveniencia, prescindimos de la notación \hat{p}_{data} y \hat{p}_{model} , podemos expresar la entropía cruzada de dos distribuciones de probabilidad p y q como

$$H(p, q) = -\sum_x p(x) \log q(x)$$

Esta expresión es similar a la definición de la divergencia KL:

$$D_{KL}(p \parallel q) = -\sum_x p(x) \log \frac{q(x)}{p(x)}$$

De dicha similitud podemos extraer una relación que conecta la entropía cruzada con las medidas de Teoría de la Información que ya conocemos. Sólo tenemos que acordarnos de la siguiente propiedad de los logaritmos $\log q/p = \log q - \log p$. Utilizando esa propiedad, la definición de la divergencia KL se descompone en

$$\begin{aligned} D_{KL}(p \parallel q) &= -\sum_x p(x) \log q(x) + \sum_x p(x) \log p(x) \\ &= H(p, q) - H(p) \end{aligned}$$

Reorganizando términos, obtenemos una expresión que define la entropía cruzada $H(p, q)$ como la suma de la entropía de p , $H(p)$, y la divergencia KL de q a p , $D_{KL}(p \parallel q)$:

$$H(p, q) = H(p) + D_{KL}(p \parallel q)$$

Esta descomposición de la entropía cruzada nos permite atribuirle un significado en términos de bits. La entropía cruzada entre dos distribuciones de probabilidad mide el número medio de bits necesario para identificar un evento de p cuando el esquema de codificación utilizado está basado en la distribución de probabilidad q en lugar de la distribución de probabilidad “real” de p . Si p corresponde a los datos observados y q al modelo que utilizamos para realizar predicciones, la entropía cruzada nos indica la cantidad de información que nos falta, en media, para predecir correctamente los valores asociados a ejemplos reales dado el modelo predictivo. La entropía cruzada, pues, nos indica el número de bits necesarios para codificar un dato cuando estamos utilizando una distribución equivocada q en vez de la distribución real p . Obviamente, cuando construyamos un modelo q intentaremos que se aproxime lo máximo posible a la distribución real p .

Intuitivamente, la entropía cruzada se puede ver como una medida de sorpresa. Nuestro modelo se construye a partir de un conjunto de datos de entrenamiento. Sin embargo, el modelo no es perfecto y sus predicciones no serán correctas para todos los datos con los que se encuentre. La entropía cruzada mide, en media, cómo de sorprendido quedará nuestro modelo cuando descubramos el valor correcto correspondiente a un dato real. Si el valor es el predicho por el modelo, nuestra sorpresa será nula. Cuando el valor no sea el esperado, la sorpresa será alta.

En el caso extremo, si un modelo cree que un valor tiene una probabilidad de ocurrencia nula, pero ese valor aparece en la práctica, la sorpresa del modelo será infinitamente grande: el modelo no tuvo en cuenta ese valor y necesita un número infinito de bits para codificarlo. En tal caso, la entropía cruzada sería infinita. Podemos evitar este problema si nos aseguramos de que el modelo nunca realice suposiciones que consideren imposible algo que puede suceder realmente (como veremos más adelante al analizar la función softmax). Desde el punto de vista numérico, eliminamos el problema si nunca evaluamos $\log 0$ (las clases negativas nunca contribuyen al error). Como estrategia defensiva de tipo preventivo, aunque no siempre resulte estrictamente necesario, también podemos limitar el valor con el que trabajamos sustituyendo $\log p$ por algo como $\log \max\{p, 1e-15\}$ para garantizar la estabilidad numérica de nuestros cálculos.

Como sabemos que tanto la entropía de Shannon como la divergencia de Kullback-Leibler toman valores no negativos, podemos deducir que la entropía cruzada será siempre mayor o igual que cero. De hecho, la entropía $H(p)$ corresponde al mínimo valor posible de la entropía cruzada $H(p, q)$: $H(p)$ es el número medio de bits necesario para codificar un ejemplo tomado de p , a los que habrá que sumar el número extra de bits necesarios cuando utilizamos una codificación basada en q en lugar de p (dados por la divergencia KL).

Cuando se comparan dos distribuciones p y q , la entropía cruzada y la entropía relativa, más conocida como divergencia KL, son idénticas salvo por una constante aditiva, la entropía de p , $H(p)$. Ambas medidas alcanzan su mínimo cuando $p = q$: $H(p, q) = H(p)$ y $D_{KL}(p \parallel q) = 0$. Si pretendemos que nuestro modelo q se parezca lo más posible a los datos p , el problema de aprendizaje lo podemos plantear como un problema de optimización en el que se pretende minimizar la divergencia KL usando el principio MDI [*Minimum Discrimination Information*] propuesto por Kullback. O, de forma completamente equivalente, en un problema de minimización de la entropía cruzada, para lo que se hace referencia al principio MCE [*Minimum Cross-Entropy*] o *Minxent*.

Cuando se entrenaen redes neuronales para resolver problemas de clasificación se suele utilizar la entropía cruzada, aunque usar la divergencia KL sería completamente equivalente. De hecho, el uso de la divergencia KL puede resultar más intuitivo dado que describe la divergencia entre el modelo y los datos. El otro término de la entropía cruzada, la entropía de los datos, es un valor fijo sobre el que no tenemos capacidad de influencia, por lo que no interviene en el proceso de optimización mediante el cual ajustamos los parámetros de la red. Dicho de otra forma, los gradientes serán los mismos si usemos la entropía cruzada o la divergencia KL. Como consecuencia, los resultados obtenidos por métodos de optimización basados en el uso del gradiente serán los mismos en ambos casos.

Si utilizamos la entropía cruzada (o la divergencia KL) como función de coste para entrenar una red neuronal, dicha función de coste tenderá a su mínimo conforme la red sea capaz de realizar mejores predicciones. Hasta aquí, exactamente igual que cuando utilizábamos el error cuadrático. No obstante, el uso de la entropía cruzada elimina la ralentización del aprendizaje que se observa cuando se emplea el error cuadrático. La tasa a la que se ajustan los pesos de la capa de salida de la red dependerá exclusivamente del error observado en la salida. Cuanto mayor sea este error, más rápido será el aprendizaje. No influirá negativamente el hecho de que las neuronas sigmoidales puedan estar operando cerca de sus zonas de saturación, como sí sucedía con el error cuadrático.

La función softmax

Cuando tenemos un problema de clasificación binaria, nos basta con entrenar una red neuronal con una sola neurona sigmoidal en la capa de salida. La salida de dicha neurona la podemos interpretar como la probabilidad de que un ejemplo dado pertenezca a la clase positiva. En ese caso, la red la podemos entrenar utilizando la entropía cruzada, tal como vimos anteriormente.

Cuando tenemos un problema de clasificación con más de dos clases, la capa de salida de nuestra red incluirá necesariamente varias neuronas.

Recuerde que la divergencia KL y la entropía cruzada no son medidas simétricas, por lo que optimizar q para que se parezca a p y optimizar p para que se parezca a q son dos problemas de minimización diferentes.

Recuerde que, con el error cuadrático, el aprendizaje era mucho más lento cuando la salida de una neurona sigmoidal estaba completamente equivocada, debido a la derivada prácticamente nula de su función de activación.

Si se trata de un problema de predicción, en el que usamos una neurona lineal como salida de la red, el error cuadrático no presenta los inconvenientes con los que nos encontramos al trabajar con neuronas sigmoidales para resolver problemas de clasificación, por lo que podemos seguir utilizándolo sin problemas.

En tal caso, podríamos seguir utilizando neuronas sigmoidales y la entropía cruzada como función de error, evaluada individualmente sobre cada neurona de salida. Habitualmente, si tenemos K clases diferentes, utilizaremos K neuronas de salida.

Si, en nuestro problema de clasificación, las clases son disjuntas, puede resultar conveniente interpretar los niveles de activación de las neuronas de salida como estimaciones realizadas por la red neuronal de la probabilidad de que un ejemplo pertenezca a cada una de las clases del problema. Para forzar que la red neuronal represente una distribución de probabilidad definida sobre las diferentes clases de nuestro problema, podemos utilizar la función *softmax*:

$$y_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

donde y_j corresponde al nivel de activación de la j -ésima neurona de salida y z_j es su entrada neta ($z_j = \text{net}_j = \sum_i w_{ij}x_i$).

La función *softmax* nos permite que la salida i -ésima de la red pueda interpretarse como una estimación de la probabilidad de que la clase i sea la clase correcta para el ejemplo correspondiente a la entrada de la red. El uso de exponentiales hace que todos los niveles de activación sean positivos, evitando de este modo el problema asociado a la sorpresa “infinita” de un modelo cuando encuentra un ejemplo de una clase que creía imposible. Esto, que daba lugar a una entropía cruzada infinita, no sucede con la función softmax, ya que e^{z_i} siempre es positivo. Además, la sumatoria del denominador nos garantiza que la suma de las salidas sea siempre uno, de forma que la salida de la red pueda interpretarse como una distribución de probabilidad.

En problemas de clasificación binaria, podemos utilizar tanto una única unidad sigmoidal de tipo logístico como una capa softmax con dos neuronas de salida. Ambas alternativas funcionarán igual de bien, si bien lo más habitual es quedarnos con una sola neurona, más simple. Cuando tengamos más de dos clases diferentes, si las clases son disjuntas, será preferible utilizar una capa softmax para que la salida de la red pueda interpretarse como una distribución de probabilidad. Si, en nuestro problema, se admite más de una respuesta correcta, como en una red que comprueba la presencia de distintos tipos de objetos en una imagen, las clases no son disjuntas. En tal caso, la salida no representa una distribución de probabilidad (las salidas no tienen que sumar 1) y debemos utilizar neuronas sigmoidales independientes para cada clase en lugar de una capa de tipo softmax.

La función softmax puede verse como una forma de reescalar los valores correspondientes a las entradas netas de las neuronas de la capa, de modo que el resultado se pueda interpretar como una distribución de probabilidad. Las entradas recibidas por la capa softmax, x_i , se

transforman mediante la matriz de pesos, w_{ij} , en un vector de entradas netas, z_j . Dichas entradas netas, en ocasiones denominadas *logits*, no están acotadas. Es la función softmax la que hace que el vector de *logits* adquiera la forma de vector de probabilidades.

A diferencia de lo que sucede habitualmente, las neuronas de la capa salida no son independientes cuando utilizamos la función softmax, motivo por el que es habitual hablar de capa softmax cuando nos referimos a la capa de salida de una red que utilice la función softmax. La función softmax introduce un componente no lineal, igual que otras funciones de activación no lineales. Además, esa activación no es local, sino que introduce un aspecto competitivo entre los niveles de activación de las distintas neuronas de la capa, a diferencia de lo que sucede con las funciones de activación que se aplican, de forma independiente, sobre neuronas individuales. Desde un punto de vista neurocientífico, es una propiedad interesante de las capas softmax, similar a la inhibición lateral que se cree que existen entre neuronas cercanas en el córtex.

¿Por qué se denomina softmax? Para entender de dónde proviene su nombre, definamos la siguiente función de activación, en la que generalizamos la función softmax introduciendo un parámetro β :

$$y_j = \frac{e^{\beta z_j}}{\sum_{k=1}^K e^{\beta z_k}}$$

donde β es una constante positiva. Cuando $\beta = 1$, tenemos la función softmax estándar. Variando el valor de la constante β , obtendremos una familia de funciones relacionadas con la función softmax.

El uso de una exponencial exagera las diferencias entre los valores z_k . El valor y_j de la función softmax será cercano a cero cuando z_j sea significativamente menor que el máximo de los valores z_k , mientras que tenderá a 1 cuando se trate del máximo, a no ser que sea extremadamente parecido al segundo mayor valor.

Conforme aumentamos el valor de β , en el límite $\beta \rightarrow \infty$, nos encontramos con que $y_j = 1$ cuando $j = \arg \max_k \{z_k\}$ y $y_j = 0$ en caso contrario. En definitiva, un vector *one-hot* correspondiente a una competición de tipo WTA [*winner takes all*]: sólo una de las salidas tiene un valor 1, mientras todas las demás quedan a 0.

Al fijar un valor concreto de β ($\beta = 1$ habitualmente), estamos utilizando una versión suavizada de una función indicadora que asigna 1 a la neurona cuya entrada neta es máxima y 0 a todas las demás. De ahí proviene el término *softmax*, aunque la función esté más relacionada realmente con la función *arg máx* que con el máximo. Tal vez debería haberse llamado *softargmax*, aunque entonces resultaría indudablemente más difícil de pronunciar.

Siguiendo con nuestra versión parametrizada de la función softmax, si utilizásemos el caso extremo $\beta \rightarrow \infty$, la capa softmax generaría una

salida basada exclusivamente en la opción aparentemente mejor (la mayor entrada neta), por pequeña que fuese la diferencia con respecto a otras alternativas posibles. Si nos vamos al extremo opuesto, $\beta = 0$, la salida de la capa softmax sería $1/K$ para todas las neuronas, lo que equivaldría a una elección completamente aleatoria entre las K alternativas de nuestro problema, que no nos aportaría información para resolver nuestro problema de clasificación. El caso intermedio, $\beta = 1$, nos permite seguir optando por la opción aparentemente más prometedora, pero sin descartar ninguna de las demás alternativas. La opción correspondiente al máximo obtendrá una porción del pastel proporcionalmente mayor que las demás, pero todas obtendrán su parte. Además, la función resultante es continua y diferenciable, dos propiedades deseables desde el punto de vista analítico que no posee la función $\arg \max$.

El uso de funciones como softmax se corresponde con la forma en la que, de forma natural, resolvemos el dilema exploración-explotación: cómo elegir entre explorar nuevas posibilidades o explotar la información de la que ya disponemos. En la práctica, no siempre se puede cuantificar el valor de la exploración, cuya recompensa final desconocemos. La solución habitual consiste en elegir inyectando algo de aleatoriedad en la selección de una opción, para asegurar una exploración adecuada de las diferentes alternativas. Al mismo tiempo, nuestra selección estocástica debe conservar cierto sesgo a favor de la opción más probable/prometedora. La función softmax proporciona una forma razonable de resolver el dilema.

La función softmax es monótona en el sentido de que la derivada parcial $\partial y_j / \partial z_k$ es positiva cuando $j = k$ y es negativa cuando $j \neq k$. Esto es, el aumento de una entrada neta z_k garantiza que la salida correspondiente a dicha entrada, y_k , también aumentará, a la vez que disminuyen todos los demás niveles de activación de las demás neuronas de la capa softmax para compensar el aumento de y_k mientras que la suma de todas las salidas sigue siendo 1.

A la hora de utilizar una capa *softmax* como capa de salida de una red neuronal entrenada con gradiente descendente y *backpropagation*, tenemos que calcular la derivada de la función softmax o, mejor dicho, las derivadas parciales de la función softmax, ya que se trata de una función vectorial. Afortunadamente, el cálculo de estas derivadas es relativamente sencillo y el resultado nos será familiar. Dada la función $\text{softmax} : \mathbb{R}^K \rightarrow \mathbb{R}^K$, hemos de calcular su matriz jacobiana, la matriz formada por sus derivadas parciales de primer orden.

$$\frac{\partial y_j}{\partial z_i} = \frac{\partial}{\partial z_i} \left(\frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \right)$$

Recordando las matemáticas de secundaria, calculamos la derivada de

un cociente

$$\left(\frac{f}{g}\right)' = \frac{f'g - g'f}{g^2}$$

donde, en nuestro caso, $f = e^{z_j}$ y $g = \sum_{k=1}^K e^{z_k}$.

Independientemente del valor de i , para la función $g = \sum_{k=1}^K e^{z_k}$, su derivada siempre será

$$\frac{\partial}{\partial z_i} \left(\sum_{k=1}^K e^{z_k} \right) = e^{z_i}$$

En el caso de la función $f = e^{z_j}$, tenemos que distinguir dos casos, en los que utilizaremos $S = \sum_{k=1}^K e^{z_k}$ para simplificar la notación:

- Cuando $i = j$, $f' = e^{z_j}$:

$$\begin{aligned} \frac{\partial y_j}{\partial z_i} &= \frac{e^{z_j} S - e^{z_i} e^{z_j}}{S^2} \\ &= \frac{e^{z_j}}{S} \frac{S - e^{z_i}}{S} \\ &= y_j(1 - y_i) \end{aligned}$$

- Cuando $i \neq j$, $f' = 0$:

$$\begin{aligned} \frac{\partial y_j}{\partial z_i} &= \frac{0 - e^{z_i} e^{z_j}}{S^2} \\ &= -\frac{e^{z_i}}{S} \frac{e^{z_j}}{S} \\ &= -y_i y_j \end{aligned}$$

En resumen, las derivadas parciales de la función softmax vienen dadas por la siguiente expresión:

$$\frac{\partial y_j}{\partial z_i} = \begin{cases} y_j(1 - y_i) & \text{si } i = j \\ -y_j y_i & \text{si } i \neq j \end{cases}$$

Para abreviar, podemos recurrir a la función delta de Kronecker:

$$\delta_{ij} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

En ocasiones, se utiliza la notación $1_{(i=j)}$ en lugar de δ_{ij} .

Su uso nos permite obtener una expresión más concisa para las derivadas parciales de la función softmax:

$$\frac{\partial y_j}{\partial z_i} = y_j (\delta_{ij} - y_i)$$

Si nos fijamos, la derivada es prácticamente idéntica a la de una neurona sigmoidal que utilice la función logística. Por este motivo, nuestra discusión anterior acerca del uso de la entropía cruzada como función de

coste sigue siendo aplicable cuando construimos una red neuronal con una capa softmax como capa de salida.

Cuando entrenamos una red neuronal, las anteriores no son las derivadas que realmente nos interesan, sino las derivadas de la salida con respecto a los parámetros de la red. Como siempre

$$\frac{\partial z_i}{\partial w_{ji}} = x_j$$

por lo que

$$\frac{\partial y_j}{\partial w_{ji}} = \frac{\partial y_j}{\partial z_i} \frac{\partial z_i}{\partial x_j} = y_j (\delta_{ij} - y_i) x_j$$

Utilizando notación vectorial, podríamos reemplazar la función delta por la matriz identidad. No obstante, en una capa softmax, se pueden calcular las derivadas de sus salidas con respecto a sus pesos sin necesidad del multiplicar matrices, aun cuando cada salida de la capa softmax depende de las entradas netas que reciben todas las neuronas de la capa.

También necesitaremos la derivadas de las salidas con respecto a las entradas, $\partial y / \partial x_i$, para poder propagar el error hacia atrás. Este cálculo, no obstante, es el mismo que se realiza en una capa completamente conectada con cualquier otra función de activación, por lo que el uso de la función softmax no introduce novedad alguna al respecto.

La función softmax se introdujo en la capa de salida de las redes neuronales multicapa utilizadas para clasificación en los años 90.⁵⁷⁴ ⁵⁷⁵ Entonces se solía denominar función sigmoidal generalizada⁵⁷⁶ o unidad de Potts.⁵⁷⁷ Su comportamiento competitivo, sin llegar a ser de tipo WTA [*winner takes all*], nos permite incorporar información relativa al problema que deseamos resolver en el entrenamiento de redes neuronales multicapa (esto es, la estimación de la distribución de probabilidad asociada a las clases de un problema de clasificación). Dado que la salida de cada neurona atempera la respuesta de las demás neuronas de salida, esta competición fomenta la cooperación entre las neuronas de una capa softmax.

Como curiosidad, cerremos esta sección mencionando que la función softmax también aparece al describir la probabilidad de que un átomo se encuentre en un nivel cuántico de energía E_i cuando el átomo forma parte de un ensemble que ha alcanzado su equilibrio térmico a temperatura T . Se trata de la distribución de Maxwell-Boltzmann, que nos da el número medio de partículas que se encuentran en cada estado de energía: $e^{-E_i/k_B T}$. Dicho valor se normaliza para que la suma para todos los niveles de energía sea 1 y obtengamos una distribución de probabilidad. En este caso, el argumento de la función softmax es la energía negativa de cada estado $-E_i$ dividida por $k_B T$.

⁵⁷⁴ John S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. En D. S. Touretzky, editor, *NIPS'1989 Advances in Neural Information Processing Systems 2*, pages 211–217. Morgan-Kaufmann, 1990a. URL <https://goo.gl/Y65Rdr>

⁵⁷⁵ John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. En Françoise Fogelman Soulié y Jeanny Hérault, editores, *Neurocomputing: Algorithms, Architectures and Applications*, volume 98 de *NATO ASI Series*, pages 227–236. Springer, Berlin, Heidelberg, 1990b. ISBN 978-3-642-76153-9. DOI: 10.1007/978-3-642-76153-9_28

⁵⁷⁶ Sridhar Narayan. The generalized sigmoid activation function: Competitive supervised learning. *Information Sciences*, 99(1):69 – 82, 1997. ISSN 0020-0255. DOI: 10.1016/S0020-0255(96)00200-9

⁵⁷⁷ David E. Rumelhart, Richard Durbin, Richard Golden, y Yves Chauvin. Backpropagation: The basic theory. En Yves Chauvin y David E. Rumelhart, editores, *Backpropagation: Theory, Architectures, and Applications*, pages 1–34. Lawrence Erlbaum Associates Inc., 1995. ISBN 0805812598

Estimación de máxima verosimilitud

En Estadística, un estimador puntual consiste en la estimación del valor de un parámetro mediante un solo valor, el que proporciona la “mejor” predicción para una cantidad que sea de nuestro interés. Ese parámetro puede ser un valor real (¿cuánto variará la cotización del dólar con respecto al euro durante el próximo año?) o un vector (el vector de parámetros de un modelo paramétrico, como puede ser el conjunto de pesos de una red neuronal). Por convención, para distinguir las estimaciones de los valores reales, se utiliza $\hat{\theta}$ para denotar la estimación de un parámetro θ .

En aprendizaje automático, partimos de un conjunto de datos de entrenamiento X y deseamos obtener una estimación $\hat{\theta}$ de los parámetros de un modelo que resulte adecuado para resolver el problema que tengamos entre manos. La estimación de máxima verosimilitud [*MLE: maximum likelihood estimation*] es un método estadístico para estimar los parámetros de un modelo dado un conjunto de observaciones. Para ello, tratamos de encontrar los parámetros del modelo que maximizan la probabilidad de realizar las observaciones de nuestro conjunto de entrenamiento dados los parámetros del modelo. El estimador de máxima verosimilitud se define como

$$\hat{\theta}_{MLE} = \arg \max_{\theta} p_{model}(X; \theta)$$

donde X representa las observaciones del conjunto de entrenamiento y θ el conjunto de parámetros del modelo. Por su parte, p_{model} estima la distribución de probabilidad real p_{data} de la que se obtuvieron los datos X del conjunto de entrenamiento. Es decir, $p_{model}(X; \theta)$ estima, para un conjunto de datos X y de parámetros θ , la probabilidad $p_{data}(X)$ de observar los datos X .

Es habitual asumir que el conjunto de m ejemplos del conjunto de entrenamiento $X = \{x^{(1)}, \dots, x^{(m)}\}$ está formado por muestras tomadas independientemente de la distribución p_{data} , por lo que podemos expresar el estimador anterior como

$$\hat{\theta}_{MLE} = \arg \max_{\theta} \prod_{i=1}^m p_{model}(x^{(i)}; \theta)$$

El uso de un producto de muchas probabilidades suele resultar problemático desde el punto de vista numérico, ya que su cálculo resulta propenso a errores de *underflow*. Por este motivo, lo habitual es utilizar el logaritmo de la verosimilitud [*log-likelihood*] en lugar de la verosimilitud. Tomar logaritmos en la expresión anterior no cambia nuestro problema de maximización y nos permite convertir el producto en una suma:

$$\hat{\theta}_{MLE} = \arg \max_{\theta} \sum_{i=1}^m \log p_{model}(x^{(i)}; \theta)$$

Si reescalamos la función que deseamos maximizar, podemos obtener una versión del criterio de máxima verosimilitud que se expresa en forma de media (esperanza, para ser precisos) con respecto a la distribución empírica \hat{p}_{data} definida por el conjunto de entrenamiento:

$$\hat{\theta}_{MLE} = \arg \max_{\theta} \frac{1}{m} \sum_{i=1}^m \log p_{model}(x^{(i)}; \theta)$$

Una forma de interpretar nuestro problema de estimación consiste en verlo como una forma de maximizar la similitud entre la distribución definida por el modelo, p_{model} , y la distribución empírica, \hat{p}_{data} , observada en forma de muestra de la distribución real de los datos, p_{data} . Maximizar la similitud, obviamente, equivale a minimizar la disimilitud. Y ya conocemos una medida que nos permite evaluar el parecido entre esas dos distribuciones: la divergencia KL $D_{KL}(\hat{p}_{data} \parallel p_{model})$. En consecuencia, podemos realizar una estimación MLE de los parámetros de un modelo minimizando la divergencia KL. El valor óptimo $\hat{\theta}$ que se obtiene es el mismo cuando maximizamos la verosimilitud y cuando minimizamos la divergencia KL.⁵⁷⁸

Si recordamos la definición de la divergencia KL,

$$D_{KL}(\hat{p}_{data} \parallel p_{model}) = H(\hat{p}_{data}, p_{model}) - H(\hat{p}_{data})$$

podemos observar que el segundo término depende sólo del conjunto de entrenamiento, por lo que minimizar la divergencia KL es equivalente a minimizar la entropía cruzada. Por tanto, minimizar la entropía cruzada es lo que hacemos cuando realizamos una estimación MLE de los parámetros de un modelo.⁵⁷⁹

Habitualmente, se dice que se emplea la entropía cruzada como función de error, coste o pérdida asociada a un problema de clasificación, especialmente en combinación con la función softmax. Técnicamente, toda función de error derivada del logaritmo de la verosimilitud es en realidad una entropía cruzada: la entropía cruzada entre la distribución empírica \hat{p}_{data} definida por el conjunto de entrenamiento y la distribución definida por el modelo p_{model} .

Para problemas de regresión

En aprendizaje supervisado, estamos interesados realmente en estimar una probabilidad condicional $p(Y|X; \theta)$ que nos permita predecir el valor de Y dado X . En este caso, el estimador MLE será de la forma:

$$\hat{\theta}_{MLE} = \arg \max_{\theta} p_{model}(Y|X; \theta)$$

Si los ejemplos $(x^{(i)}, y^{(i)})$ del conjunto de entrenamiento son independientes, la fórmula anterior se puede expresar como una suma:

$$\hat{\theta}_{MLE} = \arg \max_{\theta} \frac{1}{m} \sum_{i=1}^m \log p_{model}(y^{(i)}|x^{(i)}; \theta)$$

⁵⁷⁸ Eric B. Baum y Frank Wilczek. Supervised learning of probability distributions by neural networks. En D. Z. Anderson, editor, *NIPS'1987 Neural Information Processing Systems*, pages 52–61. American Institute of Physics, 1988. URL <https://goo.gl/v9TrQU>

⁵⁷⁹ Sara A. Solla, Esther Levin, y Michael Fleisher. Parallel Networks that Learn to Pronounce English Text. *Complex Systems*, 2(6):625–639, 1988. ISSN 0891-2513. URL <http://www.complex-systems.com/pdf/02-6-1.pdf>

Esta estimación MLE se puede utilizar para justificar, por ejemplo, por qué en problemas de regresión se utiliza el error cuadrático. El objetivo de la regresión es predecir un valor \hat{y} dado un vector de características x . En vez de asumir que nuestro modelo produce una predicción puntual \hat{y} , supongamos que nuestro modelo genera una distribución de probabilidad condicional $p(y|x)$, ya que, en un conjunto de entrenamiento real, podemos encontrarnos diferentes valores de y para los mismos valores de x . Supongamos que dicha distribución de probabilidad es una distribución de probabilidad normal $p(y|x) = \mathcal{N}(\hat{y}, \sigma^2)$ con media \hat{y} y varianza σ^2 . Entonces, el logaritmo de la verosimilitud se puede expresar como

$$\sum_{i=1}^m \log p_{model}(y^{(i)}|x^{(i)}; \theta) = -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^m \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}$$

donde $\hat{y}^{(i)}$ es la salida de nuestro modelo de regresión para la entrada $x^{(i)}$.

Dado que la varianza σ^2 la podemos ver como una constante fijada por el usuario, el término que se maximiza al utilizar el estimador MLE corresponde exactamente a la minimización del error cuadrático medio MSE:

$$MSE = \frac{1}{m} \sum_{i=1}^m \|\hat{y}^{(i)} - y^{(i)}\|^2$$

Una vez más, los dos criterios proporcionan la misma estimación de los parámetros del modelo: maximizar el estimador MLE equivale a minimizar el error cuadrático.

Para otros tipos de problemas

El estimador de máxima verosimilitud MLE puede utilizarse como criterio para definir la función de coste que resulte más adecuada para el problema particular al que nos enfrentemos, para el que podemos diseñar una red neuronal con una capa de salida a medida.

En problemas de clasificación, hemos visto que lo más razonable es utilizar la función softmax como capa de salida. En ese caso, la entropía cruzada es la función de coste idónea. En problemas de regresión, lo usual es utilizar neuronas lineales en la capa de salida y, como vimos, el error cuadrático. En general, si queremos modelar una distribución de probabilidad $p(y|x; \theta)$, el principio MLE nos sugiere que utilicemos $-\log p(y|x; \theta)$ como función de coste.

Por ejemplo, si lo que queremos es aprender la varianza σ^2 de una distribución normal en lugar de su media, como hicimos en la sección anterior, resulta conveniente parametrizar la distribución en términos de su precisión $\beta = 1/\sigma^2$, que define la anchura de la distribución:

$$\mathcal{N}_{(\mu, \beta)}(x) = \sqrt{\frac{\beta}{2\pi}} e^{-\beta(x-\mu)^2/2}$$

El logaritmo de la verosimilitud asociada a la estimación de la varianza de una gaussiana parametrizada con β quedaría como

$$-\log p(\beta|x) = -\frac{1}{2} (\log \beta - \beta(x - \mu)^2 - \log(2\pi))$$

que involucra sumas, multiplicaciones y logaritmos sobre el parámetro β , mientras que el uso de la varianza σ^2 requiere el uso de divisiones, problemáticas numéricamente para valores cercanos a cero.

Esta función de coste nos permitiría, por ejemplo, predecir una cantidad de varianza diferente para y en función del valor de x . En términos estadísticos, crear un modelo con heterocedasticidad.

Modelos más complejos basados en la combinación de distribuciones gaussianas [*GMMs: Gaussian mixture models*] también se han estudiado de forma exhaustiva. Los modelos de este tipo basados en redes neuronales se conocen como redes MDN [*mixture density networks*], cuyas salidas representan probabilidades, medias y covarianzas asociadas a las diferentes clases de un problema. Se han empleado con éxito para crear modelos generativos del lenguaje hablado⁵⁸⁰ y del movimiento de objetos físicos.⁵⁸¹ Esta estrategia permite que una red represente múltiples modas (clases) y controle sus varianzas asociadas, un aspecto crucial para la calidad de esos modelos generativos.

En resumen, para aprender la varianza de una distribución utilizaríamos la función de coste anterior, de la misma forma que utilizaríamos el error cuadrático medio MSE para aprender su media o el error absoluto medio MAE para aprender su mediana.

Una derivación alternativa

Si la sección anterior se le hizo demasiado cuesta arriba, tal vez le resulte más sencillo entender la función softmax de forma intuitiva como una generalización de las técnicas estadísticas tradicionales de regresión.

La regresión lineal es, sin duda, el modelo de regresión más utilizado. Se estima el valor \hat{y} de la variable y como una combinación lineal de las entradas, x_i , y los parámetros del modelo, θ_i :

$$\hat{y} = \theta \cdot x$$

Cuando, en lugar de un valor numérico arbitrario, lo que deseamos es predecir la probabilidad de una clase en un problema binario de clasificación, podemos introducir la función logística en nuestro modelo lineal. Como resultado, obtenemos un modelo de regresión logística:

$$\hat{y} = \sigma(\theta \cdot x) = \frac{1}{1 + e^{-\theta \cdot x}}$$

Cuando utilizamos la regresión logística, podemos interpretar su salida

⁵⁸⁰ Michael Schuster. *On Supervised Learning from Sequential Data with Applications for Speech Recognition*. PhD thesis, Nara Institute of Science and Technology, 1999

⁵⁸¹ Alex Graves. Generating sequences with recurrent neural networks. *arXiv e-prints*, arXiv:1308.0850, 2013. URL <http://arxiv.org/abs/1308.0850>

como la probabilidad de que un ejemplo pertenezca a una clase:

$$\begin{aligned} p(y = P|x) &= \hat{y} = \sigma(\theta \cdot x) \\ p(y = N|x) &= 1 - p(y = P|x) = 1 - \hat{y} = 1 - \sigma(\theta \cdot x) \end{aligned}$$

Es decir, cuando el ejemplo es de la clase positiva, deseamos que la salida de nuestro modelo de clasificación sea cercana a 1. Del mismo modo, ante un ejemplo de la clase negativa, la salida debería ser próxima a 0. De forma que asignaremos el valor deseado $t = 1$ a los ejemplos de la clase positiva y $t = 0$ a los ejemplos de la clase negativa. Una forma de combinar ambos requisitos en una única expresión consiste en definir

$$p(y|x) = \hat{y}^t(1 - \hat{y})^{1-t}$$

Cuando $t = 1$, seguimos teniendo $p(y|x) = \hat{y}$ y, cuando $t = 0$, $p(y|x) = 1 - \hat{y}$. Si tomamos logaritmos en la expresión anterior, obtenemos una expresión familiar:

$$\log p(y|x) = t \log \hat{y} + (1 - t) \log(1 - \hat{y})$$

Una vez más, llegamos a la conclusión de que minimizar el error en nuestro problema de clasificación es equivalente a minimizar la entropía cruzada. Al menos, para problemas binarios de clasificación.

¿Cómo se generaliza la regresión logística para problemas de clasificación con más de dos clases? En Estadística, se emplea la regresión logística multinomial. No debería sorprendernos a estas alturas que la regresión logística multinomial también se conozca por el nombre de regresión logística multiclasa o politómica, regresión softmax, logit multinomial o clasificador de máxima entropía.

Una forma de llegar al modelo multinomial es imaginar que, para un problema de K clases, se ejecutan $K - 1$ modelos de regresión logística independientes. Una de las clases (p.ej. la clase K) se elige como pivote y se hacen $K - 1$ regresiones de las restantes clases con respecto al pivote:

$$\begin{aligned} \log \frac{p(y = 1)}{p(y = K)} &= \theta_1 \cdot x \\ \log \frac{p(y = 2)}{p(y = K)} &= \theta_2 \cdot x \\ &\dots \\ \log \frac{p(y = K - 1)}{p(y = K)} &= \theta_{K-1} \cdot x \end{aligned}$$

donde cada uno de los $K - 1$ modelos de regresión contiene un vector completo de parámetros θ_i .

Si exponenciamos en ambos lados de las expresiones anteriores, pode-

La función de pérdida que se minimiza realmente es, obviamente, la media de las entropías cruzadas para todos los ejemplos del conjunto de entrenamiento.

mos obtener las probabilidades asociadas a cada clase:

$$\begin{aligned} p(y = 1) &= p(y = K) e^{\theta_1 \cdot x} \\ p(y = 2) &= p(y = K) e^{\theta_2 \cdot x} \\ &\dots \\ p(y = K - 1) &= p(y = K) e^{\theta_{K-1} \cdot x} \end{aligned}$$

Dado que todas las probabilidades han de sumar 1, tenemos que

$$p(y = K) = 1 - \sum_{k=1}^{K-1} p(y = k) = 1 - \sum_{k=1}^{K-1} p(y = K) e^{\theta_k \cdot x}$$

de donde se deduce que

$$p(y = K) = \frac{1}{1 + \sum_{k=1}^{K-1} e^{\theta_k \cdot x}}$$

Finalmente, obtenemos las probabilidades para las demás clases:

$$p(y = j) = \frac{e^{\theta_j \cdot x}}{1 + \sum_{k=1}^{K-1} e^{\theta_k \cdot x}}$$

De esta forma, generalizamos la regresión logística para problemas multiclase y obtenemos una función que representa la distribución de probabilidad asociada a las diferentes clases del problema: la función softmax.

La función softmax también se puede derivar de forma natural de un modelo log-lineal. Esto es, se modela el logaritmo de la probabilidad de una clase como un modelo lineal al que se añade un término adicional de normalización, el logaritmo de la función de partición:

$$\log p(y = j) = \theta_j \cdot x - \log Z$$

donde Z se elige para garantizar que las probabilidades sumen 1:

$$Z = \sum_{k=1}^K e^{\theta_j \cdot x}$$

Las ecuaciones resultantes para las probabilidades de las distintas clases corresponden, como es lógico, a la función softmax:

$$p(y = j) = \frac{e^{\theta_j \cdot x}}{1 + \sum_{k=1}^K e^{\theta_k \cdot x}}$$

De esta interpretación de la regresión logística proviene el término *logit*, con el que se denominan los logaritmo de las razones relativas, de oportunidades, de probabilidades, de disparidad, de exceso o de “momios” [*odds ratios*, en inglés] con las que se pueden interpretar los parámetros de los modelos logísticos en términos de apuestas.

Observe que antes sólo utilizábamos $K - 1$ vectores de parámetros, mientras que ahora volvemos a utilizar K vectores de parámetros. Formalmente, este modelo softmax está “sobreparametrizado”, lo que quiere decir que para cualquier modelo que ajustemos a los datos existen múltiples configuraciones de parámetros que dan exactamente los mismos resultados: si le restamos un vector fijo ψ a los K vectores de parámetros θ_k , obtenemos exactamente los mismos resultados, como puede comprobar fácilmente. Siempre podemos eliminar el vector de parámetros θ_K haciendo $\psi = \theta_K$, de forma que se reemplace θ_K por $\theta_K - \psi = 0$.

La derivación log-lineal de la función softmax también nos facilita interpretar el comportamiento de la función softmax. Dado que $\log p(y = j) = z_j - \log Z$ y $Z = \sum e^z$, tenemos que $\log p(y = j) = z_j - \log \sum e^z$, lo que se puede aproximar por $\log p(y = j) \approx z_j - \max z$.

En realidad, no resulta estrictamente necesario utilizar la función softmax, sino que podríamos haber utilizado cualquier otra función no negativa en lugar de la exponencial o restringir las entradas netas a las neuronas de salida para que no puedan ser negativas. Para obtener una distribución de probabilidad, basta con que, al final, dividamos por la suma. Los estadísticos lo denominan modelo BTL [Bradley-Terry-Luce].

Cuestiones de implementación

A la hora de implementar una red neuronal que utilice la función softmax en su capa de salida para resolver problemas de clasificación, hemos de tener cuidado con algunos detalles.

Estabilidad numérica

La implementación directa de la fórmula correspondiente a la función softmax podría tener un aspecto similar al siguiente:

```
function y = softmax(z)
    y = exp(z) / sum(exp(z))
```

No obstante, esta implementación resulta problemática. Al utilizar el código anterior para calcular la función softmax, nos podemos llevar una pequeña sorpresa:

```
softmax([1 2 3]) = [ 0.09 0.24 0.67 ]
softmax([101 102 103]) = [ 0.09 0.24 0.67 ]
softmax([1001 1002 1003]) = [ NaN NaN NaN ]
```

En el primer caso, el código se ejecuta correctamente y obtenemos el resultado deseado. Lo mismo sucede en el segundo caso. No obstante, en el tercer caso, que podría darse perfectamente en la práctica dado que las entradas netas de las neuronas de una capa softmax no están acotadas, se produce un error de desbordamiento al calcular e^z , lo que da lugar a una indeterminación `NaN` cuando se divide `+Inf/+Inf`. Aun cuando no se hubiese producido un desbordamiento, la división de números grandes en un ordenador es numéricamente inestable, motivo por el que utilizaremos un truco de normalización.

En el ejemplo anterior se puede observar una propiedad interesante de la función softmax: $\text{softmax}(z) = \text{softmax}(z + D)$, siendo D un escalar

que añadimos a nuestro vector de entradas z . Partiendo de la expresión de la función softmax, podemos multiplicar numerador y denominador por una constante C :

$$\frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} = \frac{Ce^{z_j}}{C \sum_{k=1}^K e^{z_k}} = \frac{Ce^{z_j}}{\sum_{k=1}^K Ce^{z_k}} = \frac{e^{z_j + \log C}}{\sum_{k=1}^K e^{z_k + \log C}}$$

de tal forma que, si hacemos $D = \log C$, obtenemos la propiedad antes mencionada

$$\text{softmax}(z) = \text{softmax}(z + D)$$

Tenemos libertad absoluta a la hora de elegir el valor de D sin que cambie el resultado de la evaluación de la función softmax sobre un vector de entradas netas z . Esto nos permite seleccionar un valor D que mejore la estabilidad numérica del cálculo de la función softmax. Habitualmente, se escoge $D = -\max z_k$, de forma que el valor más alto que recibe como argumento la función softmax sea siempre 0. En la práctica, para calcular la función softmax aplicamos la siguiente fórmula:

$$\text{softmax}(z) = \text{softmax}(z - \max_k z_k)$$

En pseudocódigo, esto se traduce en algo como

```
function y = softmax(z)
    y = exp(z-max(z)) / sum(exp(z-max(z)))
```

Usando esta implementación revisada, el máximo valor para el que se calcula la exponencial es $z_{\max} = 0$, por lo que ninguno de los términos que aparecen en la expresión será nunca mayor que 1. Con esta implementación, que es numéricamente estable, se consigue siempre el resultado correcto al evaluar la función softmax:

```
softmax([1 2 3]) = [ 0.09 0.24 0.67 ]
softmax([101 102 103]) = [ 0.09 0.24 0.67 ]
softmax([1001 1002 1003]) = [ 0.09 0.24 0.67 ]
```

Nuestra implementación revisada hace que los valores de z sean cercanos a cero (si no existen diferencias demasiado grandes entre ellos). Salvo el correspondiente al máximo, todos los valores serán negativos. Al exponentiarlos, los valores negativos de mayor magnitud dan lugar a un error de *underflow*. Por ejemplo,

```
softmax([1000 2000 3000]) = [ 0.00 0.00 1.00 ]
```

Aunque el resultado siga siendo numéricamente incorrecto, ya que la auténtica función softmax nunca devuelve cero. La aparición de ceros debidos a errores de *underflow* es menos problemática que la aparición

de infinitos e indefiniciones `NaN` de la implementación original. Como uno de los términos de la sumatoria del denominador siempre valdrá 1 (e^0), nunca dividiremos por un valor cercano a cero, de forma que nunca se producirán divisiones por cero ni errores de desbordamiento [*overflow*]. Así pues, la versión revisada de nuestra implementación nos permite evaluar la función softmax con un error numérico reducido, aun cuando el vector z incluya números muy grandes.

El único problema de nuestra implementación revisada se produce si, en algún momento, queremos calcular el logaritmo del resultado de la función softmax. Errores de *underflow* en el numerador hacen que la expresión pueda evaluarse como cero. Si, para calcular `log(softmax(z))` primero llamamos a la función `softmax(z)` y luego calculamos el logaritmo, erróneamente podríamos obtener `-Inf` como resultado. Tendremos que implementar, de forma independiente, una función `logSoftmax(z)` que sea numéricamente estable, igual que hicimos con la función `softmax(z)`.

A la hora de realizar nuestra implementación de una capa softmax, también podemos acordarnos de que la función softmax está sobreparametrizada. Dado que la suma de las salidas, interpretables como una distribución de probabilidad, tiene que ser siempre 1, en realidad nos basta con ajustar $K - 1$ vectores de parámetros. Siempre que no usemos algún tipo de regularización como *weight decay* (regularización L2), lo que nos obligaría a tratar todos los pesos de la red de forma uniforme, podemos elegir una de las neuronas de salida y dejarla desconectada de la red (esto es, su entrada neta será siempre cero). Mantener conectadas todas las neuronas hace que el conjunto de pesos sea redundante y podría ralentizar el entrenamiento de la red. Es una consecuencia más de que múltiples valores de entrada diferentes produzcan exactamente la misma salida para la función softmax: $\text{softmax}(z) = \text{softmax}(z + D)$. Por otro lado, mantener todas las neuronas conectadas puede suponer una ventaja que hace que resulte más fácil de entrenar la red neuronal. Que exista una sola combinación óptima de parámetros puede hacer más difícil de resolver el problema de optimización. Dado que en una capa softmax existen múltiples configuraciones equivalentes y sólo nos interesan los valores relativos de sus parámetros, tal vez eso pueda contribuir a que un algoritmo basado en el gradiente descendente obtenga mejores resultados.

La función de coste

Aunque el error cuadrático medio, MSE, sea la función de coste más popular y utilizada, dado que corresponde a la estimación MLE de los pesos de la red que resulta óptima para predecir la media de distribuciones gaussianas de probabilidad,⁵⁸² la función de coste más adecuada para problemas de clasificación hemos visto que es la basada en la divergencia KL o, de modo completamente equivalente, en la entropía cruzada.

Algunas bibliotecas, como Theano, están diseñadas para detectar y estabilizar expresiones numéricamente inestables de forma automática. Si nuestras bibliotecas no incorporan esa funcionalidad, tendremos que ser nosotros los que nos encarguemos de garantizar la corrección numérica de los cálculos que realicemos.

⁵⁸² David E. Rumelhart, Richard Durbin, Richard Golden, y Yves Chauvin. Backpropagation: The basic theory. En Yves Chauvin y David E. Rumelhart, editores, *Backpropagation: Theory, Architectures, and Applications*, pages 1–34. Lawrence Erlbaum Associates Inc., 1995. ISBN 0805812598

Cuando se entrena una red para resolver un problema de clasificación, ya sea minimizando el error cuadrático medio MSE o la entropía cruzada, sus salidas aproximan las “probabilidades” a posteriori de pertenencia de un ejemplo a una clase. “Probabilidades”, entre comillas, porque esta estimación dependerá de forma directa de la regularización a la que sometamos el entrenamiento de la red. En el límite, si utilizamos *weight decay* (regularización L2) con un factor de regularización muy elevado, todos los pesos de la red tomarán valores muy pequeños y las probabilidades de salida serán casi uniformes. En un caso no tan extremo, las distribuciones de probabilidad tendrán picos más marcados y, con conjuntos de entrenamiento enormes, tenderán a producir soluciones óptimas en sentido bayesiano. Aunque, tal vez, deberíamos hablar de confianzas más que de probabilidades en sentido estricto: el orden relativo de las salidas es interpretable, mientras que sus valores absolutos (o sus diferencias) técnicamente no lo son.

En cualquier caso, hemos de ser conscientes de que el uso de una función de error u otra no implica necesariamente la minimización de la tasa de error de la red en la práctica. El entrenamiento de la red puede conducir a soluciones subóptimas (por ejemplo, debidas a zonas planas en el espacio de pesos que evitan que un algoritmo de optimización basado en el gradiente funcione correctamente).

En realidad, el uso de una u otra función de coste depende de la interpretación subjetiva que hagamos de los vectores de salida de la red. Por ejemplo, Richard Golden mostró, ya en 1988, que minimizar el error cuadrático es equivalente a realizar una estimación MLE si asumimos que los vectores de salida, tanto los observados como los deseados, corresponden a los picos de distribuciones gaussianas de probabilidad.⁵⁸³ No existe, desde el punto de vista formal, una función de error “correcta”.

¿Por qué decimos entonces que resulta más adecuada la función basada en la entropía cruzada? Simplemente, porque evita el problema que se produce cuando una neurona sigmoidal se satura produciendo una salida errónea. La derivada de su función de activación será muy baja, motivo por el que necesitamos una función de error con una pendiente muy pronunciada que facilite un cambio razonable en los pesos de la neurona durante su entrenamiento. Cuando una neurona que debería activarse (tener salida 1) pasa de una salida de 0.000000001 a una salida de 0.000001, una diferencia inferior a una millonésima, su error cuadrático cambia muy ligeramente, de forma prácticamente inapreciable. Sin embargo, su entropía cruzada disminuye mucho, ya que se ha producido una mejora notable en la salida de la neurona. De hecho, como ya vimos, la escarpada pendiente de la entropía cruzada compensa de forma exacta la planitud de una función sigmoidal saturada.

Gracias a la pronunciada pendiente de la entropía cruzada, se pueden observar mejoras notables en la velocidad del aprendizaje de una red

⁵⁸³ Richard M. Golden. A unified framework for connectionist systems. *Biological Cybernetics*, 59(2): 109–120, 1988. ISSN 1432-0770. DOI: 10.1007/BF00317773

neuronal cuando se utiliza la entropía cruzada como función de coste.⁵⁸⁴ En términos del error absoluto observado, el uso de MSE tiende a producir mayores errores relativos para valores pequeños de salida. En cambio, la entropía cruzada ayuda a estimar de forma más precisa esos valores pequeños.^{585,586,587}

Si, durante el entrenamiento de una red neuronal, comenzamos con una tasa de aprendizaje demasiado elevada, es relativamente sencillo que los pesos de las neuronas adquieran valores positivos muy altos o negativos muy bajos. Esto fomentará que las neuronas de la capa de salida se saturen. En el caso tradicional, esto hará que el gradiente del error en las capas ocultas de la red sea minúsculo. Como consecuencia, la desaparición del gradiente hará que seamos incapaces de corregir adecuadamente los valores erróneos de los pesos de la red y el error de la red no disminuirá. La saturación de las neuronas de la capa salida hará que nos quedemos estancados en una aparente meseta [*plateau*], que resulta fácil de confundir con un mínimo local. El uso de una función de coste basada en la entropía cruzada mitiga este problema, al menos en la parte que corresponde a la saturación de las neuronas sigmoidales de salida.

La función de error, coste o pérdida basada en la entropía cruzada se calcula, para un lote de n ejemplos, como sigue:

$$E_{CE} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^K t_{ij} \log y_{ij}$$

donde t_{ij} es la salida deseada para la j -ésima neurona de la capa softmax dado el i -ésimo ejemplo del conjunto de datos y y_{ij} es la salida observada en la j -ésima neurona para el i -ésimo ejemplo.

Para un ejemplo de la clase k , utilizaremos como salida deseada un vector *one-hot* de la forma $t = (0, \dots, 0, 1, 0, \dots, 0)$ en el que $t_k = 1$ para la clase k y $t_j = 0$ para todas las demás. Esto es, el único elemento del vector t diferente de cero será el elemento que marca la clase correcta k para el ejemplo que estemos clasificando. Si denominamos k a la clase correcta correspondiente al ejemplo x de nuestro conjunto de datos, podemos simplificar la fórmula que nos permite calcular la entropía cruzada:

$$E_{CE} = -\frac{1}{n} \sum_{i=1}^n t_{ik_i} \log y_{ik_i} = -\frac{1}{n} \sum_{i=1}^n \log y_{ik_i}$$

El valor de la función de coste para un ejemplo de la clase k sólo depende de la salida observada en la k -ésima neurona, la correspondiente a la clase del ejemplo. Así pues, nuestra función de error no es más que $-\log P(\text{data}|\text{model})$. Estrictamente, se trata del negativo del logaritmo de una verosimilitud. No es una entropía cruzada en sí, sino la media de las entropías cruzadas de los ejemplos individuales del conjunto de

⁵⁸⁴ Kiyotoshi Matsuoka y Jianqiang Yi. Backpropagation based on the logarithmic error function and elimination of local minima. En *Proceedings of the 1991 IEEE International Joint Conference on Neural Networks*, pages 1117–1122 vol.2, 1991. DOI: 10.1109/IJCNN.1991.170546

⁵⁸⁵ Sara A. Solla, Esther Levin, y Michael Fleisher. Parallel Networks that Learn to Pronounce English Text. *Complex Systems*, 2(6):625–639, 1988. ISSN 0891-2513. URL <http://www.complex-systems.com/pdf/02-6-1.pdf>

⁵⁸⁶ Geoffrey E. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 40(1):185–234, 1989. ISSN 0004-3702. DOI: 10.1016/0004-3702(89)90049-0

⁵⁸⁷ Herbert Gish. A probabilistic approach to the understanding and training of neural network classifiers. En *ICASSP'1990 International Conference on Acoustics, Speech, and Signal Processing*, pages 1361–1364 vol.3, 1990. DOI: 10.1109/ICASSP.1990.115636

datos (interpretando t_i e y_i como distribuciones de probabilidad binarias $(t_i, 1 - t_i)$ e $(y_i, 1 - y_i)$, respectivamente).

Aparentemente, la función de coste no penaliza los falsos positivos de forma directa. Sin embargo, dado que la función softmax es, en realidad, una versión suavizada de la función arg máx, la presencia de un falso positivo hará que disminuya el valor de salida asociado a la clase correcta y aumente la contribución de ese ejemplo al error ECE . Si se utilizase esta función de coste para una capa de salida formada por neuronas independientes, entonces no se estarían penalizando los falsos positivos.

Que elijamos la función de error adecuada, obviamente, no quiere decir que desaparezcan todos los problemas de entrenamiento de la red neuronal. En redes utilizadas para resolver problemas de clasificación, ya usen el error cuadrático o la entropía cruzada, es relativamente sencillo que la red aprenda rápidamente a activar cada neurona de salida en proporción a la probabilidad de la clase correspondiente. Una vez que la red encuentra esta estrategia, puede resultar complicado reducir el error de la red utilizando los ejemplos del conjunto de entrenamiento. Desde el punto de vista de la función de coste, se ha llegado a una meseta de la que puede resultar difícil salir (y, erróneamente, podemos confundir esta meseta con un mínimo local de la función de coste).

En resumen, para construir un clasificador basado en redes neuronales, lo más recomendable es utilizar una función de error derivada de la entropía cruzada, ya que la entropía cruzada se considera preferible al error cuadrático medio. Dado que el uso de una función de error u otra afecta al cálculo del gradiente, el entrenamiento de la red se realizará de forma diferente en función de la función de error escogida. Una vez entrenada la red, no obstante, su efectividad se evaluará en términos de su tasa de error, como cualquier otro modelo de clasificación. Por desgracia, ninguna función de coste nos garantiza que se vaya a minimizar ese error de clasificación.

El gradiente de la función de coste

A la hora de implementar una capa de tipo softmax para que pueda entrenarse con algoritmos basados en el gradiente descendente y *back-propagation*, hemos de calcular las derivadas del error con respecto a las entradas netas de las neuronas. Ese gradiente estará formado por las derivadas parciales del error con respecto a cada entrada neta z_j . Cuando analizamos el algoritmo de propagación de errores, llamábamos deltas a esas derivadas. El cálculo de los deltas nos permitirá luego propagar el error hacia atrás (hacia las entradas de la capa) y calcular las derivadas del error con respecto a los pesos de la red (para ajustar sus valores utilizando el gradiente descendente).

El uso de una función de activación concreta de la capa de salida de

una red va normalmente asociada a la utilización de una función de coste particular. En ocasiones, el cálculo de la derivada del error con respecto a los distintos parámetros de la red resulta muy sencillo. Es el caso de la derivada de la función de coste basada en la entropía cruzada cuando utilizamos una capa softmax. Calculemos esa derivada con respecto a las entradas netas z_j de las neuronas de la capa:

$$\begin{aligned}\frac{\partial E_{CE}}{\partial z_j} &= -\sum_{k=1}^K \frac{\partial t_k \log(y_k)}{\partial z_j} = -\sum_{k=1}^K t_k \frac{\partial \log(y_k)}{\partial z_j} = -\sum_{k=1}^K t_k \frac{1}{y_k} \frac{\partial y_k}{\partial z_j} \\ &= -\frac{t_j}{y_j} \frac{\partial y_j}{\partial z_j} - \sum_{k \neq j} t_k \frac{\partial y_k}{\partial z_j} = -\frac{t_j}{y_j} y_j (1 - y_j) - \sum_{k \neq j} \frac{t_j}{y_k} (-y_k y_j) \\ &= -t_j + t_j y_j + \sum_{k \neq j} t_k y_j = -t_j + \sum_{k=1}^K t_k y_j = -t_j + y_j \sum_{k=1}^K t_k \\ &= y_j - t_j\end{aligned}$$

Tras algunos cálculos, en los que tenemos que distinguir $\partial y_k / \partial z_j$ para $k = j$ y para $k \neq j$, llegamos al resultado final $\partial E_{CE} / \partial z_j = y_j - t_j$, el mismo que para una neurona lineal. Y también el mismo que correspondía a la derivada de la entropía cruzada para una única neurona sigmoidal y que nos permitía evitar la ralentización del aprendizaje de la red neuronal. El cálculo de este gradiente resulta muy conveniente, tanto por su simplicidad como por su estabilidad numérica.

Este gradiente nos permitirá utilizar el algoritmo de propagación de errores en redes cuyas capas de salida utilicen la función softmax (y una función de coste basada en el logaritmo de la verosimilitud, la entropía cruzada).

La capa softmax

Llegados a este punto, ya disponemos de todos los ingredientes necesarios para implementar una capa softmax con la que construir redes neuronales que nos permitan resolver problemas de clasificación. Como hicimos al estudiar el algoritmo de propagación de errores, proporcionaremos una implementación modular que se pueda combinar fácilmente con otros tipos de capas para construir redes neuronales más complejas.

Nuestra capa softmax será una capa completamente conectada (todas las entradas se conectan a todas las salidas) e implementará la versión sobreparametrizada de la función softmax. Por tanto, la podemos definir como una subclase (un tipo particular) de `FullyConnectedLayer`:

```
class SoftmaxLayer(n,m): FullyConnectedLayer(n,m)
    function y = forward(x);
    function backward(error);
```

El método `forward` es el encargado de propagar una señal de entrada `x` para obtener una salida `y`. Su implementación será muy similar a la de una capa completamente conectada, ya que sólo cambia la función de activación.

```
class SoftmaxLayer(n,m): FullyConnectedLayer(n,m)
    function y = forward(x)
        for j=1:m
            z = b[j];
            for i=1:n
                z += w[j][i]*x[i];
            m = max(z);
            y = exp(z-m);
            s = sum(y);
            y = y/s;
```

Observe cómo, en el fragmento de código anterior, hemos implementado la versión numéricamente estable de la función softmax.

El método `backward` será el encargado de propagar una señal de error hacia atrás y generar todos los “deltas” necesarios. En este caso, tenemos que calcular el gradiente del error con respecto a los pesos `w` (`deltaW`) y con respecto a las entradas `x` (`deltaX`). Ambos gradientes los calculamos a partir del gradiente con respecto a la entrada neta de la neurona, nuestro `delta` (`dE/dz`). La implementación de la capa softmax se completa tal como aparece a continuación:

```
class SoftmaxLayer(n,m): FullyConnectedLayer(n,m)
    function backward(error)
        // dE/dz = dE/dy * dy/dz
        for j=1:m
            delta[j] = 0;
            for k=1:m
                if (k==j)
                    delta[j] += error[k] * y[k] * (1-y[j]);
                else
                    delta[j] += error[k] * y[k] * -y[j];
        // dE/dw (con respecto a los pesos w)
        for j=1:m
            for i=1:n
                deltaW[j][i] = delta[j] * x[i];
        // dE/dx (con respecto a las entradas x)
        for i=1:n
            deltaX[i] = 0;
            for j=1:m
                deltaX[i] += delta[j] * w[j][i];
```

En este caso, la implementación modular nos obliga a calcular de forma explícita la derivada de la función softmax. Tal como describimos en la sección anterior, esta implementación se puede simplificar si integramos el cálculo de la derivada del error con respecto a la salida (la señal *error* que proviene del exterior, dE/dy) con el de la derivada de la salida con respecto a la entrada neta (la derivada de la función de activación, dy/dz). Entonces, nuestro delta sería, simplemente, $\text{delta}[j] = y[j] - \text{target}[j]$, que equivale al gradiente del error cuando usamos el error cuadrático como función de coste.

Para terminar esta sección, mostramos cómo se implementa una capa softmax en una GPU. Utilizando CUDA y la biblioteca CuDNN proporcionada por NVidia, la implementación eficiente de una capa softmax se realiza de la siguiente manera:

```
class CUDASoftmaxLayer(n,m): SoftmaxLayer(n,m)

    function y = forward(x)
        cublasSgemm(1,w,x,0,z); // z = wx
        cudnnSoftmaxForward(1,z,0,y);

    function backward(error)
        cudnnSoftmaxBackward(1,y,error,0,delta);
        // deltaW = x * delta
        cublasSgemm(1,x,delta,0,deltaW);
        // deltaX = w * delta
        cublasSgemm(1,w,delta,0,deltaX);
```

La implementación es muy similar a la de una capa convencional. Sólo hemos cambiado el uso de funciones de activación por llamadas a funciones específicamente diseñadas para trabajar con capas softmax (`cudnnSoftmaxForward` y `cudnnSoftmaxBackward`), además de prescindir de las operaciones relacionadas con los sesgos de las neuronas, que no intervienen en nuestra capa softmax.

Inicialización de los pesos

La inicialización de los pesos de una capa softmax se realiza de forma aleatoria, como en cualquier otra capa de una red neuronal multicapa.

Aunque la implementación descrita anteriormente no incluye sesgos en las neuronas correspondientes a la capa softmax, se podrían introducir sin problemas. En tal caso, inicializar los sesgos a cero suele ser la estrategia más habitual, al ser compatible con la mayoría de las estrategias de inicialización de los pesos.

No obstante, cuando se trata de una capa de salida, puede resultar beneficioso inicializar los sesgos de tal forma que, ante la ausencia de otras entradas, se reproduzca algún estadístico de los valores de salida. En

el caso de problemas de clasificación como los que resolvemos utilizando capas softmax de salida, podríamos inicializar los sesgos de las neuronas de la capa softmax para que su salida reproduzca la distribución de clases del problema de clasificación para el que estamos diseñando la red neuronal.

A la hora de fijar los valores iniciales de los sesgos b , asumimos que el resto de los pesos iniciales son lo suficientemente pequeños como para que la salida de la red venga determinada exclusivamente por los sesgos de las neuronas.

$$\text{softmax}(b) = p_{\text{data}}$$

Para conseguir el efecto deseado, sólo tenemos que calcular la función de activación inversa aplicada a la distribución de clases del problema y utilizar los valores que obtengamos para inicializar los sesgos de la capa softmax. En el caso particular de la función softmax, basta con inicializar los pesos con $b = -\log p_{\text{data}}$, donde p_{data} representa la distribución de clases del problema (la probabilidad de que un ejemplo pertenezca a cada una de las clases en el conjunto de datos de entrenamiento).

Esta estrategia también es válida para otros tipos de redes neuronales, como *autoencoders* o máquinas de Boltzmann, en las que nos interesa que su salida sea parecida a su entrada.

Técnicas de regularización

En cuanto a las técnicas de regularización que se suelen emplear para prevenir el sobreaprendizaje, podemos seguir utilizándolas cuando empleamos capas softmax. Sólo hemos de tener en cuenta algunos matices:

- *Weight decay*

El aprendizaje de un clasificador softmax nunca llega a converger, dado que la función softmax nunca llega a predecir una probabilidad 1 (ni, obviamente, 0) para una clase particular. Esto puede hacer que la red continúe aprendiendo, cada vez con pesos más grandes, para intentar realizar predicciones más extremas. Esto se puede prevenir utilizando alguna estrategia de regularización de los pesos, como la consistente en regularizar la función de coste utilizando *weight decay* (regularización L2). Eso sí, en caso de que utilicemos alguna técnica de regularización que opere sobre los pesos, hemos de garantizar que todos los pesos se traten de forma uniforme, lo que nos obliga a utilizar la versión sobreparametrizada de la función softmax (esto es, K vectores de pesos para las K clases del problema).

- *Suavizado de las salidas [label smoothing]*

Cualquier conjunto de datos con el que trabajemos puede contener errores, también en las etiquetas que nos indican la clase asociada a cada ejemplo. Podemos asumir que la etiqueta y asociada a un ejemplo de entrenamiento es correcta sólo con probabilidad $1 - \epsilon$, siendo ϵ una constante pequeña. Entonces, podemos incorporar esta suposición en

el entrenamiento de la red sin necesidad de añadir ruido a nuestro conjunto de datos de forma explícita. En el caso de un problema de clasificación con K clases, sólo tenemos que reemplazar los objetivos 0 y 1 en nuestro vector de salidas deseadas por los objetivos $\epsilon/(K-1)$ y $1-\epsilon$, respectivamente. Podemos aplicar directamente las funciones de coste estándar, como la entropía cruzada, sobre sobre esas salidas suavizadas para incorporar la incertidumbre ϵ en el entrenamiento de nuestro modelo de clasificación. Se trata de una sencilla estrategia que se viene utilizando habitualmente desde los años 80.⁵⁸⁸

■ *Dropout*

Si echamos la vista atrás, podemos recordar que *dropout* era una técnica de regularización popular en *deep learning* que consistía en entrenar un único modelo de tal forma que dicho modelo fuese equivalente a disponer de un ensemble completo. *Dropout* lo conseguía suprimiendo aleatoriamente enlaces en la red, de forma que, entrenando una única red, conseguíamos las ventajas que nos ofrecería entrenar una familia enorme de redes con la que construir un ensemble.

Cuando se utiliza una red neuronal entrenada con *dropout* para clasificar, se utilizan todas las neuronas de la red multiplicando sus salidas por la probabilidad p de que se incluya cada neurona en uno de los modelos del ensemble [*weight scaling inference rule*]. Esta estrategia de evaluación de la salida de un ensemble a partir de un único modelo es exacta para las capas softmax, en las que $y = \text{softmax}(z) = \text{softmax}(w \cdot x + b)$. En este caso, la salida de cada modelo del ensemble se obtendría multiplicando, elemento a elemento, la entrada x por un vector binario d :

$$y_d = \text{softmax}(w \cdot (d \odot x) + b)$$

La predicción del ensemble se obtendría renormalizando las probabilidades predichas por cada modelo individual, para lo que se debe utilizar una media geométrica:

$$\tilde{y}_{\text{ensemble}} = \frac{\tilde{y}_{\text{ensemble}}}{\sum \tilde{y}_{\text{ensemble}}}$$

donde

$$\tilde{y}_{\text{ensemble}} = \sqrt[2^n]{\prod_{d \in \{0,1\}^n} y_d}$$

Para comprobar que la regla de inferencia utilizada en *dropout* es exacta para una capa softmax, simplificamos $\tilde{y}_{\text{ensemble}}$:

⁵⁸⁸ Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, y Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *arXiv e-prints*, arXiv:1512.00567, 2015b. URL <http://arxiv.org/abs/1512.00567>

$$\begin{aligned}
\tilde{y}_{ensemble} &= \sqrt[2^n]{\prod_{d \in \{0,1\}^n} softmax(w \cdot (d \odot x) + b)} \\
&= \sqrt[2^n]{\prod_{d \in \{0,1\}^n} \frac{e^{w \cdot (d \odot x) + b}}{\sum e^{w \cdot (d \odot x) + b}}} \\
&= \frac{\sqrt[2^n]{\prod_{d \in \{0,1\}^n} e^{w \cdot (d \odot x) + b}}}{\sqrt[2^n]{\prod_{d \in \{0,1\}^n} \sum e^{w \cdot (d \odot x) + b}}}
\end{aligned}$$

Como $\tilde{y}_{ensemble}$ será normalizada para producir una distribución de probabilidad, podemos ignorar la constante del denominador:

$$\begin{aligned}
\tilde{y}_{ensemble} &\propto \sqrt[2^n]{\prod_{d \in \{0,1\}^n} e^{w \cdot (d \odot x) + b}} \\
&= e^{\frac{1}{2^n} \sum_{d \in \{0,1\}^n} e^{w \cdot (d \odot x) + b}} \\
&= e^{\frac{1}{2} w \cdot (d \odot x) + b}
\end{aligned}$$

Por tanto, la predicción del ensemble equivale a clasificar con una capa softmax que utilice una matriz de pesos $w/2$, donde w representa los pesos obtenidos durante el entrenamiento con *dropout* usando $p = 1/2$.

Tenga en cuenta que el resultado es exacto sólo cuando se utiliza una única capa de tipo softmax y no se puede extraer a redes profundas con componentes no lineales en sus capas ocultas, pese a que funcione bien con ellas en la práctica.

Aplicaciones

La aplicación más común de la función softmax en redes neuronales es su utilización como capa de salida de un clasificador. Entrenando la red con una función de coste basada en la entropía cruzada, se obtiene de esa forma una extensión no lineal de la regresión logística multinomial.

- *Softmax jerárquico*

Cuando los problemas de clasificación tienen un número elevado de clases, puede resultar útil utilizar una versión jerárquica de la función softmax.⁵⁸⁹ El softmax jerárquico descompone las etiquetas asociadas a las clases en forma de árbol, de forma que cada etiqueta corresponda a un camino desde la raíz hasta una hoja de ese árbol. Entonces, se entrena un clasificador softmax para cada nodo del árbol, cuya función es elegir entre seguir por su rama izquierda o por su rama derecha.

La estructura que definamos para el árbol puede afectar notablemente al rendimiento del softmax jerárquico y es dependiente del problema particular que queremos resolver. El softmax jerárquico puede ser útil cuando las clases corresponden a palabras de un idioma como el español o el inglés (caso en el que podemos aprovechar la estructura de una ontología como WordNet para dar estructura a

⁵⁸⁹ Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, y Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *arXiv e-prints*, arXiv:1310.4546, 2013b. URL <http://arxiv.org/abs/1310.4546>

nuestro árbol). También se puede utilizar con éxito para identificar objetos en imágenes (por ejemplo, la base de datos ImageNet, usada en competiciones de visión artificial, incluye más de veinte mil categorías diferentes).

- *Softmax para problemas de regresión*

Hay quien sugiere que la función softmax puede resultar útil para resolver problemas de regresión (sí, ha leído bien). La idea es que ajustar de forma precisa los parámetros de un modelo de regresión con unidades lineales en su capa de salida es mucho más delicado que ajustar los pesos de una capa softmax, que ya sabemos que ofrece el mismo resultado para múltiples configuraciones de sus parámetros al ser $\text{softmax}(z) = \text{softmax}(z + D)$ para cualquier escalar D . En una capa softmax no importan tanto los valores particulares de los pesos sino sus magnitudes relativas. El entrenamiento de una capa softmax es menos sensible a la presencia de *outliers* que los modelos de regresión entrenados con el error cuadrático como función de coste, para el que los *outliers* desvirtúan el cálculo del gradiente. Además, algunas técnicas de regularización como *dropout* son más frágiles cuando se utiliza el error cuadrático para crear un modelo de regresión basado en redes neuronales.

Con la función softmax podemos afrontar un problema de regresión discretizando los valores de salida; es decir, cuantizándolos en intervalos adecuados para nuestro problema particular. Por ejemplo, en un problema de *rating* como el que resuelven los sistemas de recomendación para predecir las evaluaciones de un usuario en una escala de una a cinco estrellas, podemos construir un clasificador softmax en lugar de un modelo de regresión tradicional.

Como beneficio adicional, la función softmax nos ofrece una distribución de probabilidad sobre las posibles salidas del modelo, en lugar de la simple estimación puntual de la salida de un modelo de regresión, que no aporta indicaciones adicionales acerca de la confianza de la predicción realizada.

- *Softmax en las capas ocultas de la red*

Aunque pueda parecer lo contrario, el uso de la función softmax no se limita a las capas de salida de las redes neuronales. Hay, al menos, dos situaciones en las que se ha utilizado con éxito en las capas ocultas de la red.

La primera de ellas es en el entrenamiento de redes RBF normalizadas. Las redes RBF son redes neuronales con una sola capa oculta que utilizan funciones de base radial [*RBF: radial basis function*] como funciones de activación en su capa oculta y una capa lineal de salida. La normalización de la salida de la capa oculta en una red RBF da lugar a una red RBF normalizada. Dicha normalización suele traducirse

en mejoras de la precisión de la red.

La segunda es mucho más reciente y corresponde a la implementación de mecanismos de atención en redes neuronales artificiales, que analizaremos a continuación.

Mecanismos de atención

Normalmente, no somos realmente conscientes de la mayor parte de lo que debería ser obvio para nuestros sentidos.⁵⁹⁰ Aunque podamos pensar que estamos observando una escena como si nuestro cerebro recibiese directamente una imagen en alta definición, realmente no es así. En función de en qué nos estemos fijando (o hayamos sido predispuestos a fijarnos), pasaremos por alto multitud de detalles. Sólo cuando desviamos nuestra atención hacia fragmentos particulares de una escena somos capaces de darnos cuenta de cómo encajan las pequeñas piezas del puzzle. En ocasiones, nuestra atención se desvía por motivos naturales, probablemente derivados de nuestra evolución como especie. Como un acto reflejo, nos fijamos en un movimiento inesperado, que podría obligarnos a tomar medidas inmediatas. En otras ocasiones, nuestros intereses particulares del momento guían nuestra atención, buscando en la imagen la información que deseamos obtener (por eso puede resultar tan difícil recordar detalles concretos de una imagen si no estábamos predispuestos a observarlos en el momento en que percibimos la imagen). Y, por supuesto, hay quien sabe jugar con nuestros recursos limitados de atención para que no nos percatemos de un truco, el recurso habitual de cualquier mago. No sólo nuestra percepción del mundo es una construcción mental que no representa fielmente la realidad (algo que ya comentamos al analizar distintos tipos de ilusiones visuales) sino que, además, tenemos la falsa impresión de percibir una imagen completa y detallada cuando sólo vemos lo que necesitamos ver y no más.

Tal vez esté pensando: “Y todo esto, ¿qué tiene que ver con las redes neuronales artificiales?”. En la práctica, pueden surgir problemas en los que nos interese dotar a una red neuronal artificial de un mecanismo similar a nuestra atención.

- *Descripción textual de imágenes*^{591,592,593}

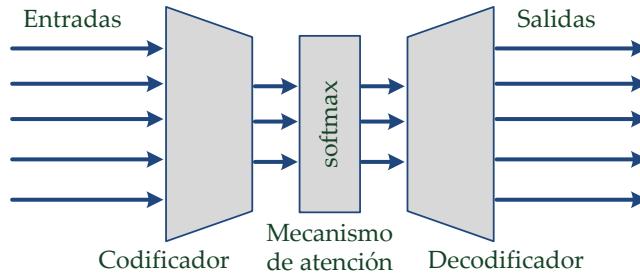
Puede que nos interese ir variando los aspectos concretos de una imagen en los que nos fijamos en cada momento, en función de qué queramos hacer. A diferencia de nuestra percepción visual, la percepción de una imagen en una red neuronal artificial sí se representa de forma estática. Si suministramos una imagen como entrada a una red neuronal, la red construirá una representación abstracta de la imagen de entrada como un conjunto de niveles de activación en las neuronas de sus capas ocultas. Si, por ejemplo, queremos construir una red que construya una

⁵⁹⁰ David Eagleman. *Incognito: The Secret Lives of the Brain*. Pantheon, 2011. ISBN 0307377334

⁵⁹¹ Oriol Vinyals, Alexander Toshev, Samy Bengio, y Dumitru Erhan. Show and Tell: A Neural Image Caption Generator. *arXiv e-prints*, arXiv:1411.4555, 2014. URL <http://arxiv.org/abs/1411.4555>

⁵⁹² Kyunghyun Cho, Aaron C. Courville, y Yoshua Bengio. Describing multimedia content using attention-based encoder-decoder networks. *arXiv e-prints*, arXiv:1507.01053, 2015. URL <http://arxiv.org/abs/1507.01053>

⁵⁹³ Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, y Yoshua Bengio. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *CoRR*, abs/1502.03044, 2015. URL <http://arxiv.org/abs/1502.03044>



representación textual de la imagen, nos resultará muy útil ser capaces de implementar un mecanismo de atención que vaya centrándose en distintos aspectos de la imagen conforme va generando un texto descriptivo.

- *Traducción automática*⁵⁹⁴

El uso de mecanismos de atención puede ser clave si queremos construir una red neuronal que traduzca textos de un idioma a otro. En un sistema de traducción automática, cuando estamos generando una nueva palabra de salida, en cierto modo estamos fijándonos en una palabra particular de la secuencia de palabras de entrada. Dado que el orden de las palabras en una frase no es el mismo en todos los idiomas, nos resultará extremadamente útil disponer de un mecanismo que nos permita centrarnos en cada momento en la palabra del texto original que sea más relevante para generar la siguiente palabra del texto traducido.

- *Reconocimiento de voz*⁵⁹⁵

Los mecanismos de atención también se han utilizado para mejorar el rendimiento de los sistemas de reconocimiento de voz, en los que la entrada es una señal acústica y la salida es la representación textual de dicha señal.

Tanto la descripción textual de imágenes y vídeos como la traducción automática se pueden resolver combinando dos redes neuronales en una arquitectura conocida como codificador-decodificador [*encoder-decoder*]. La primera de las redes, el codificador, recibe una entrada y la representa en forma de vector de activaciones de sus neuronas. Esta representación vectorial se utiliza como contexto a partir del cual la segunda red, el decodificador, genera la salida deseada. Esto es, la salida del codificador sirve de entrada para el decodificador.

Si, para nuestro problema particular, queremos que la salida del decodificador vaya cambiando en función de la parte del contexto que sea más relevante en cada momento, sólo tenemos que intercalar una función softmax entre el codificador y el decodificador. El mecanismo de atención se implementa en forma de capa softmax cuya salida se aplica a la salida del codificador para modelar la entrada que, en cada momento,

Figura 148: Implementación de mecanismos de atención con la función softmax en una red codificador-decodificador.

⁵⁹⁴ Dzmitry Bahdanau, Kyunghyun Cho, y Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *ICLR’2015*, arXiv:1409.0473, 2015. URL <http://arxiv.org/abs/1409.0473>

⁵⁹⁵ Jan Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, KyungHyun Cho, y Yoshua Bengio. Attention-based models for speech recognition. *arXiv e-prints*, arXiv:1506.07503, 2015. URL <http://arxiv.org/abs/1506.07503>

recibe el decodificador.

La incorporación de mecanismos de atención en redes neuronales puede utilizarse para, en una arquitectura codificador-decodificador, extraer de forma automática las correspondencias entre dos modalidades diferentes de una señal (imagen y texto descriptivo, clip de vídeo y descripción verbal, señal de voz y su representación textual, texto en un idioma y el mismo texto traducido a otro idioma).

Técnicas similares a los mecanismos de atención aquí descritos pueden utilizarse como mecanismos de direccionamiento para acceder a la memoria externa de redes con memoria [*memory networks*]⁵⁹⁶ o máquinas de Turing neuronales [*Neural Turing Machines*].⁵⁹⁷

Apéndice: El gradiente natural

El gradiente descendente estocástico, que se ha convertido en el algoritmo estándar para el entrenamiento una red neuronal artificial en *deep learning*, se basa en tomar muestras del gradiente de la función de error, coste o pérdida tras cada actualización de los parámetros de la red. Repitiendo el mismo proceso una y otra vez esperamos alcanzar un mínimo de dicha función de coste (idealmente, un mínimo global, aunque nunca tendremos garantías de ello).

Existe una forma alternativa de interpretar el proceso de optimización que se utiliza para entrenar los parámetros de una red neuronal. ¿En qué consiste esta interpretación del entrenamiento de la red? Básicamente, en ver la red como una distribución de probabilidad de sus valores de salida dada una entrada particular.

Esta interpretación alternativa requiere que entendamos qué es la divergencia KL, motivo por el que su discusión la hemos retrasado hasta este capítulo. Como recordará, la divergencia KL es una medida de cómo se parece una distribución de probabilidad a otra. Cuanto mayores sean las diferencias entre las dos distribuciones, mayor será su divergencia KL. Obviamente, para dos redes con los mismos parámetros, la divergencia KL de sus salidas será 0 (siempre darán la misma salida para cada entrada particular). Cuando las redes tengan diferentes parámetros, no obstante, las salidas obtenidas por ambas para una misma entrada variarán, por lo que su divergencia KL será mayor que 0. Cuanto mayores sean las diferencias en las probabilidades de salida, mayor será la divergencia KL entre ambas redes.

Cuando actualizamos ciegamente los parámetros de una red de acuerdo al gradiente de la función de coste, no tenemos garantía alguna de que la distribución de salida de la nueva red sea similar a la de la red antes de actualizar sus parámetros. Esto nos lleva, de forma natural, a la idea del gradiente natural (valga la redundancia).⁵⁹⁸

El gradiente natural añade una restricción adicional a las actualizacio-

⁵⁹⁶ Sainbayar Sukhbaatar, arthur szlam, Jason Weston, y Rob Fergus. End-to-end memory networks. En C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, y R. Garnett, editores, *Advances in Neural Information Processing Systems 28*, pages 2440–2448. Curran Associates, Inc., 2015. URL <https://goo.gl/PQRALU>

⁵⁹⁷ Alex Graves, Greg Wayne, y Ivo Danihelka. Neural Turing Machines. *arXiv e-prints*, arXiv:1410.5401, 2014. URL <http://arxiv.org/abs/1410.5401>

⁵⁹⁸ Shun ichi Amari. Neural Learning in Structured Parameter Spaces - Natural Riemannian Gradient. En M. C. Mozer, M. I. Jordan, y T. Petsche, editores, *NIPS'1996 Advances in Neural Information Processing Systems 9*, pages 127–133. MIT Press, 1997. URL <https://goo.gl/y4jKF8>

nes de parámetros que se realizan al entrenar la red. Obligamos a que la red con los pesos actualizados se comporte de forma similar a la red que teníamos antes de actualizar sus pesos.

Para evitar que la actualización de los pesos de una red haga cambiar de forma drástica su distribución de salida, consideramos las diferentes combinaciones de parámetros que han lugar a una red cuya divergencia KL sólo difiera de la original en una constante. Esta constante puede verse entonces como una tasa de aprendizaje. De todas esas combinaciones que conservan la divergencia KL, elegiremos aquella que minimiza nuestra función de pérdida. La tasa de aprendizaje, en vez de estar ligada a los valores de los parámetros de la red, vendrá ligada a la distribución de la salida de la red.

Como beneficio, conseguimos que el proceso de aprendizaje sea más estable. Especialmente cuando utilizamos el gradiente descendente estocástico, en el que cada estimación del gradiente se deriva de una muestra aleatoria del conjunto de datos y la presencia de anomalías en esa muestra podría inducir cambios drásticos en los parámetros de la red. Con el gradiente descendente natural, una sola actualización nunca puede afectar demasiado al comportamiento de la red.

Las actualizaciones basadas en el gradiente natural se basan en la divergencia KL, que sólo tiene en cuenta las salidas de la red. No influye cómo modelemos la red, ya que el gradiente natural será el mismo utilizaremos la función de activación que utilicemos (mientras que el gradiente convencional de la señal cambia en cuanto modificamos la función de activación de las neuronas de la red).

La idea clave del gradiente natural es medir cómo le afecta a la salida de la red un cambio de sus parámetros. Al usar una tasa de aprendizaje que no se emplea para la determinar el tamaño del cambio al que sometemos cada parámetro, el gradiente natural consigue que no todos los parámetros de la red se traten de forma uniforme. La escala asociada al cambio al que sometemos cada parámetro se ajusta, de forma automática, en función de cómo afecta a la salida de la red. El cálculo es más costoso que si usamos el gradiente convencional, aunque elimina muchos de los hiperparámetros que tenemos que ajustar al entrenar una red neuronal artificial.

El gradiente natural⁵⁹⁹ puede ayudarnos a interpretar algunas de las técnicas de optimización que se han propuesto para entrenar modelos de *deep learning*, además puede servir de fuente de inspiración para nuevos algoritmos de entrenamiento de redes neuronales. Por ejemplo, el método de optimización de políticas basado en regiones de confianza^{600,601} [*TRPO: Trust Region Policy Optimization*] es una variante del gradiente natural que se ha utilizado para conseguir que una red neuronal aprenda a jugar a videojuegos Atari usando los píxeles de las imágenes de la pantalla como entrada.

⁵⁹⁹ Razvan Pascanu y Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv e-prints*, arXiv:1301.3584, 2014. URL <http://arxiv.org/abs/1301.3584>

⁶⁰⁰ John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, y Pieter Abbeel. Trust Region Policy Optimization. *arXiv*, arXiv:1502.05477, 2015b. URL <http://arxiv.org/abs/1502.05477>

⁶⁰¹ John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, y Philipp Moritz. Trust region policy optimization. En *Proceedings of the 32nd International Conference on Machine Learning*, pages 1889–1897. PMLR, 2015a. URL <http://proceedings.mlr.press/v37/schulman15.html>

¿Cómo funcionan internamente estas técnicas de optimización?

En lugar de utilizar una función de error como el error cuadrático medio o la entropía cruzada y propagar el error, como hacemos habitualmente al emplear el gradiente descendente y *backpropagation*, se define una métrica de distancia basada en la divergencia KL (cuánto difiere la distribución actualizada de la original). Obtenemos una matriz de tamaño $n \times n$ para una red de n parámetros que nos permite calcular la distancia de un vector con respecto a alguna métrica `metric`. Para calcular la distancia asociada a un cambio `delta` en los parámetros de la red, evaluamos la siguiente función:

```
function d = distance (delta, metric)
    d = 0;
    for i=1:n
        for j=1:n
            d += delta[i]*delta[i]*metric[i][j]
```

Si nuestra matriz `metric` fuese la matriz identidad, nuestra medida de distancia equivale a la distancia euclídea. Sin embargo, no utilizaremos la matriz identidad, sino la matriz correspondiente a la medida de información de Fisher, la que contiene las segundas derivadas de la divergencia KL. Esta matriz nos permite medir la distancia de los `delta` en términos de la divergencia KL, de tal forma que evaluamos los cambios en los parámetros, en términos del gradiente de la función de error, sino del impacto que tiene cada uno de ellos sobre la salida de la red.

La matriz de Fisher nos permite tener en cuenta las relaciones existentes entre los distintos parámetros de la red. Su inversa nos permite determinar automáticamente cómo han de escalarse los cambios sobre cada uno de los distintos parámetros de la red (igual que usábamos la inversa de la matriz hessiana en el método de Newton). Si ∇f es el gradiente de nuestra función de coste y F es la matriz de Fisher, el gradiente natural $\tilde{\nabla}f$ viene dado por

$$\tilde{\nabla}f = F^{-1}\nabla f$$

Como es lógico, si usamos como matriz de Fisher la matriz identidad, el gradiente natural será el gradiente convencional. En realidad, sólo estamos cambiando la función objetivo de nuestro problema de optimización.

El cálculo de la inversa de la matriz de Fisher puede resultar computacionalmente costoso, motivo por el que implementaciones de técnicas como TRPO utilizan gradientes conjugados para encontrar el valor del gradiente natural $\tilde{\nabla}f$ que satisface la ecuación $\tilde{\nabla}f * F = \nabla f$.

El método de Newton es exactamente igual que el gradiente descendente basado en el gradiente natural, con la matriz hessiana H reemplazando la métrica F . La diferencia estriba en que la matriz hessiana de una función es local, diferente para cada punto, mientras que se suele asumir que la métrica es constante.

Redes convolutivas

Si existe un tipo particular de red neuronal artificial que marca la diferencia en la práctica, ése es precisamente el correspondiente a las redes que se utilizan para procesar señales: las redes convolutivas. Su éxito en la resolución de problemas de visión artificial que, hasta hace poco, se consideraban casi intratables, sirvió para volver a poner de moda las redes neuronales en Inteligencia Artificial. Su explotación comercial dio lugar a lo que hoy entendemos por deep learning.

Tal como su propio nombre indica, las redes convolutivas se basan en el uso de convoluciones, una operación matemática familiar para todo el que haya trabajado en procesamiento digital de señales. Normalmente, se hace referencia a este tipo de redes mediante el acrónimo inglés CNN [Convolutional Neural Network] o, simplemente, con la composición acróstica ConvNets.

Una señal es, en principio, cualquier cantidad física detectable mediante la que se pueden transmitir mensajes. Puede ser una señal de tráfico, la luz de un semáforo, el piloto parpadeante de un coche que va a cambiar de dirección o el gesto de un árbitro señalando el punto de penalty entre las protestas del público en un partido de fútbol. En general, podemos interpretar una señal como una variable que contiene información relevante sobre algún sistema de interés para nosotros. Dicha señal puede ser una función de otras variables. En el caso más sencillo, una señal x puede representar una cantidad que varía a lo largo del tiempo, por lo que podemos representarla mediante la función $x(t)$. Es lo que sucede, por ejemplo, cuando utilizamos un micrófono para captar un sonido. También podemos trabajar con señales que dependen de varias dimensiones. El ejemplo más habitual, una imagen bidimensional que podemos representar mediante una función $f(x, y)$, donde x e y corresponden a las coordenadas espaciales que nos permiten consultar el valor de la imagen en un punto concreto.

Las señales, definidas como funciones, pueden ser continuas o discretas. Las señales continuas pueden resultar convenientes desde el punto de vista matemático al tratarse de funciones de variable real (univariadas en el caso de una señal de voz, multivariadas en el caso de las imágenes). Sin embargo, dado que trabajamos con ordenadores digitales, los sensores utilizados para captar las señales las discretizan, por lo que siempre trabajaremos con señales discretizadas. Una señal discreta está formada por una serie de muestras de la señal. En el caso de señales de tipo

temporal, el número de muestras vendrá dado por la frecuencia de muestreo (p.ej. 44.1kHz en señales de audio digital, dado que el oído humano tiene un rango de 20Hz a 20kHz y el teorema de muestreo de Nyquist-Shannon nos indica que la frecuencia de muestreo debe ser el doble de la frecuencia máxima de la señal que uno quiere reproducir). En el caso de las imágenes, el número de muestras determinará la resolución de la imagen en píxeles (p.ej. 1920x1080 en una imagen de alta definición usando una relación de aspecto 16 : 9 entre el ancho y el alto de la imagen).

Las redes convolutivas son las redes neuronales artificiales que se utilizan habitualmente para resolver múltiples problemas prácticos que requieren procesar imágenes. Por ejemplo, cuando la cámara frontal de un vehículo autónomo capta una señal de tráfico, debe identificar de qué señal concreta se trata (clasificación de imágenes). También puede interesarnos detectar qué tipos de objetos aparecen en la imagen correspondiente a una escena y localizarlos dentro de la imagen (detección de objetos). Incluso podemos utilizar redes convolutivas como parte de un sistema que genere una descripción textual del contenido de una imagen, con lo que podemos indexar imágenes para realizar búsquedas por contenido en bases de datos de imágenes o sintetizar una señal de voz a partir de la descripción textual para usuarios invidentes.

¿En qué se diferencian las redes convolutivas de las redes multicapa que ya hemos estudiado con detalle? Principalmente, en que tanto sus entradas como sus salidas pueden ser estructuradas. Las redes convolutivas nos permitirán aprovechar dicha estructura para diseñar arquitecturas especializadas que resuelvan de un modo más eficiente problemas que trabajen con tipos particulares de señales.

En lugar de recibir un vector de entradas correspondientes a diferentes variables (cuyas relaciones entre ellas desconocemos), recibiremos como entrada un vector (1D), matriz (2D) o tensor (>2 D) en el que podemos explotar la relación física existente entre las diferentes entradas. En el caso de señales unidimensionales, puede tratarse de una señal de audio en el que entradas adyacentes corresponden a muestras consecutivas en el tiempo. En el caso de señales bidimensionales, las entradas pueden corresponder a los píxeles de una imagen captada con una cámara. También podemos tener señales bidimensionales de audio, como las provenientes de un array de micrófonos. En ocasiones, las señales de entrada puede que tengan más de dos dimensiones, como las imágenes en color (en la que tenemos imágenes individuales para distintos canales, como rojo, verde y azul), los datos volumétricos de imágenes médicas obtenidas mediante un escáner de tomografía computerizada o una resonancia magnética, o un simple vídeo (una secuencia de imágenes de dos dimensiones que incorpora también una dimensión temporal). De hecho, los vídeos en color serían señales de cuatro dimensiones: las dos correspondientes a las coordenadas

La relación de aspecto tradicional, 4 : 3, proviene del estándar fijado por William Dickson y Thomas Alva Edison en 1892 para cintas de 35mm. Dicho estándar imita el campo de visión del ojo humano: 155° en horizontal y 120° en vertical. Con la popularización de la televisión, la asistencia a salas de cine disminuyó y la industria del cine creó pantallas anchas para diferenciarse de la televisión, de donde proviene la relación 16 : 9 ($4^2 : 3^2$).

Dispositivos como Kinect, de Microsoft, incorporan un array de micrófonos para facilitar la separación de señales y captar correctamente una señal de voz individual en una habitación en la que pueden existir otros ruidos, entre ellos los generados por la propia consola de videojuegos

dentro de un fotograma, la correspondiente al canal de vídeo (rojo, verde o azul en una codificación RGB) y la correspondiente al tiempo. Aunque sonidos e imágenes, estáticas o en movimiento, sean los tipos de señales más habituales, existen otros muchos tipos de entradas cuya estructura podemos aprovechar usando redes convolutivas, desde el análisis de series temporales (definidas sobre cualquier cantidad que varíe en el tiempo) hasta el modelado de movimientos (p.ej. los cambios de posición de los puntos de referencia utilizados para modelar el movimiento del esqueleto de un personaje en una película de animación).

La salida de una red convolutiva puede ser convencional, para resolver problemas de clasificación usando una capa softmax o problemas de regresión usando unidades lineales. No obstante, habrá situaciones en las que su salida también reflejará la estructura del problema que pretendemos resolver. Si trabajamos con imágenes, podemos construir una red convolutiva cuya salida sea también una imagen. Esta imagen de salida, por ejemplo, nos permitirá etiquetar píxeles individuales de la imagen de entrada para marcar las zonas de la imagen en las que aparece algo que resulte de nuestro interés.

Orígenes: El Neocognitrón de Fukushima

Los orígenes de las redes convolutivas actuales hay que buscarlos en los modelos de redes neuronales artificiales que se propusieron, ya en los años 70, basándose en los estudios de David Hubel y Torsten Wiesel sobre el córtex visual. Hubel y Wiesel estudiaron el córtex estriado, la primera parte del córtex visual que interviene en el procesamiento de la información visual, la que recibe información directamente del tálamo (el relé intermedio entre la retina y el córtex visual). Los trabajos de Hubel y Wiesel, por los que recibirían el Premio Nobel en 1981, se inspiraron en los estudios sobre la retina de Stephen Kuffler y la descripción de la organización columnar del córtex somatosensorial de Vernon Mountcastle, investigaciones que se llevaron a cabo en los años 50 del siglo XX. En los años 70, Hubel y Wiesel dieron forma a su modelo del ‘cubito de hielo’ [*ice cube model*], un modelo de procesamiento de la información visual en el córtex organizado por columnas. Originalmente, sugirieron que cada pequeño fragmento del córtex, al que llamaron “hipercolumna”, contiene un conjunto completo de elementos de procesamiento de imágenes. Con el tiempo, conforme se fueron conociendo más detalles sobre diferentes tipos de neuronas, su conectividad y la organización del sistema visual, comenzaron a identificarse las limitaciones de la idea general de que el procesamiento de información visual es estrictamente jerárquico y del modelo original de Hubel y Wiesel en particular.

En la misma década en la que Hubel y Wiesel publicaron su modelo sobre la posible organización del córtex visual, un investigador japonés,

Kuhiniko Fukushima, propuso un modelo de red neuronal artificial claramente bioinspirado: el cognitrón.⁶⁰² Unos años después, publicaría una versión mejorada de su modelo: el neocognitrón.⁶⁰³ Según los modelos de esta época, la señal captada por conos y bastones en la retina se transmite hasta el núcleo geniculado lateral en el tálamo [*LGN: Lateral Geniculate Nucleus*]. Las neuronas del LGN tienen un campo receptivo circular; esto es, cada una de ellas capta las señales de una pequeña región de la imagen, de forma circular. Del tálamo, la señal visual pasa al córtex visual, que se modela de forma jerárquica. Esta arquitectura jerárquica acabaría en las neuronas abuela [*grandmother cells*], células individuales que responden a estímulos visuales complejos (como la correspondiente a una fotografía de nuestra entrañable abuela).

En los modelos de Fukushima,⁶⁰⁴ la red neuronal está compuesta por una retina, que sirve de capa de entrada y una jerarquía de módulos o niveles. Cada módulo o nivel está compuesto de dos capas: una de células simples [*S-cells: simple cells*] y otra de células complejas [*C-cells: complex cells*]. A su vez, las capas se dividen en planos (arrays de elementos):

- Las células S del mismo plano detectan la misma característica pero en diferentes localizaciones de la entrada (comparten parámetros usando *weight sharing*, de acuerdo a la terminología actual).
- Los distintos planos permiten detectar características diferentes en la entrada. El número de planos C es menor o igual que el planos S, lo que permite que se combinen características similares de planos S.
- Las células C integran las respuestas de grupos de células S (de un mismo plano en la capa S o de distintos planos).
- En la última capa de la red, el campo receptivo efectivo de cada célula es la retina completa; esto es, sólo hay una célula por plano (las neuronas “abuela”).

Como veremos más adelante, la arquitectura de las redes de Fukushima es muy similar a las de las redes convolutivas actuales: las células S corresponderían a las capas convolutivas de las redes actuales, mientras que las células C desempeñarían un rol similar al de las capas de *pooling* o submuestreo. La principal diferencia entre ambos modelos reside en que los modelos de Fukushima carecían de un algoritmo de entrenamiento adecuado. En el caso de las redes convolutivas, se utiliza la misma estrategia que en las redes multicapa convencionales: gradiente descendente y propagación de errores hacia atrás. En el caso del cognitrón o del neocognitrón, su mecanismo de “entrenamiento” es algo más laborioso. El término “entrenamiento” aparece entrecomillado porque la red, más que entrenarse como cualquier modelo de aprendizaje automático, se diseña manualmente a medida. Veamos cómo:

⁶⁰² Kunihiro Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20(3): 121–136, 1975. ISSN 1432-0770. DOI: 10.1007/BF00342633

⁶⁰³ Kunihiro Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4): 193–202, 1980. ISSN 1432-0770. DOI: 10.1007/BF00344251

⁶⁰⁴ Kunihiro Fukushima. A neural network for visual pattern recognition. *Computer*, 21(3):65–75, March 1988. ISSN 0018-9162. DOI: 10.1109/2.32

No es necesario que haya el mismo número de planos de *S-cells* y de *C-cells*, ni que los planos tengan el mismo tamaño en diferentes capas, incluso dentro del mismo módulo/nivel.

■ Capas S

En cada capa S se incluye un plano V_C con el mismo número de neuronas que el resto de planos de la capa S. Este plano proporciona una señal inhibitoria que se controla con un parámetro de selectividad r_l que regula la activación de las neuronas S (la fuerza relativa de la señal excitatoria con respecto a la señal inhibitoria que se necesita para que la salida de la neurona no sea cero). Las salidas de las neuronas S, además, se rectifican como en las unidades ReLU.

Los pesos del plano V_C no se entran, sino que se utilizan valores preestablecidos con el objetivo de favorecer la detección de patrones centrados en el campo receptivo de la neurona S.

En cuanto a los pesos del resto de los planos de una capa S, se utiliza un mecanismo de aprendizaje no supervisado de tipo competitivo. Los pesos de la célula S con mayor respuesta se incrementan paulatinamente, igual que el peso asociado a su conexión de inhibición con el plano V_C , de forma similar al aprendizaje Hebbiano. Obsérvese que los pesos sólo crecen, de forma ilimitada, aunque la salida de la neurona se mantiene acotada por la definición de su función de activación.

El mecanismo de aprendizaje no supervisado permite que, con el tiempo, cada plano comience a responder a una característica particular de su entrada y menos a otras características, de forma que los distintos planos se van especializando en identificar características concretas.

También existe la posibilidad de prescindir del mecanismo de aprendizaje no supervisado y diseñar manualmente los pesos adecuados para la resolución de un problema práctico concreto, como puede ser el reconocimiento óptico de caracteres [OCR: *Optical Character Recognition*].⁶⁰⁵

■ Capas C

Igual que en las capas S, las capas C incluyen un plano de unidades inhibitorias V_C que están conectadas a todos los planos S de la capa anterior. Su misión es que sólo las células cuyo nivel de excitación supere la media respondan con una salida positiva. El resto de los planos de la capa C pueden estar conectados a uno o varios planos de células S de la capa anterior.

Los parámetros asociados a las células C se fijan manualmente en función de la aplicación concreta del cognitrón/neocognitrón. En ocasiones, todos los planos de la misma capa C comparten exactamente los mismos pesos. Por ejemplo, se pueden utilizar valores uniformes para obtener un promedio de los valores recibidos de la capa S.⁶⁰⁶

Fukushima propuso diferentes modelos de red neuronal artificial. Algunas versiones del neocognitrón, por ejemplo, incorporan conexiones de realimentación y mecanismos de inhibición lateral que no estaban

⁶⁰⁵ Kunihiko Fukushima, Sei Miyake, y Takayuki Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):826–834, September/October 1983. ISSN 0018-9472. DOI: 10.1109/TSMC.1983.6313076

⁶⁰⁶ Murali M. Menon y Karl G. Heinemann. Classification of patterns using a self-organizing neural network. *Neural Networks*, 1(3):201 – 215, 1988. ISSN 0893-6080. DOI: 10.1016/0893-6080(88)90026-3. URL <http://www.sciencedirect.com/science/article/pii/0893608088900263>

presentes en el neocognitrón original. Los mecanismos de inhibición lateral sirven para evitar ambigüedades cuando varias unidades se activan en paralelo, como en las imágenes ambiguas que se utilizan a modo de ilusiones visuales. La realimentación se emplea para fomentar la aparición de resonancia como en ART [*Adaptive Resonance Theory*], otro de los modelos populares de redes neuronales artificiales de la época. Esta realimentación, en principio, sería la que nos permitiría poder llegar a realizar interpretaciones alternativas de una misma imagen.

El principal inconveniente del neocognitrón, no obstante, es que no se dispone de un algoritmo de entrenamiento genérico que pueda utilizarse para ajustar automáticamente todos los parámetros de la red. Las redes convolutivas, utilizando una arquitectura similar más sencilla, se pueden entrenar utilizando el gradiente descendente y *backpropagation*. Sus orígenes se remontan a los trabajos realizados por Yann LeCun a finales de los años 80, cuando, tras trabajar como investigador postdoctoral en el laboratorio de Geoffrey Hinton en la Universidad de Toronto,^{607,608} pasó a formar parte del Departamento de Investigación en Sistemas Adaptativos de los Laboratorios Bell de la AT&T en Holmdel, Nueva Jersey.

Durante los años 90, LeCun fue perfeccionando las redes convolutivas para resolver problemas de reconocimiento de dígitos manuscritos y construir sistemas OCR.^{609,610} En la década de los 90, el número de capas utilizadas en las redes convolutivas era limitado, principalmente por las limitaciones computacionales de los ordenadores de la época. Pese a tales limitaciones, las primeras redes profundas entrenadas con gradiente descendente y *backpropagation* son de esa época. El ejemplo más conocido es LeNet, el sistema diseñado por LeCun (de ahí su nombre) para leer códigos postales o dígitos manuscritos en cheques de banco.⁶¹¹ Dicho sistema llegó a ser responsable del procesamiento automático del 10 % de los cheques emitidos en Estados Unidos, un país en el que es habitual pagar los recibos habituales de agua o electricidad enviando un cheque por correo postal.

De forma completamente independiente al trabajo de LeCun en la Universidad de Toronto y AT&T, un equipo de investigadores del Centro Médico de la Universidad de Georgetown desarrollaron las redes convolutivas partiendo del neocognitrón de Fukushima. De hecho, puede que fuesen ellos los primeros en denominarlas así: redes neuronales convolutivas CNN [*Convolution Neural Networks* entonces, *Convolutional Neural Networks* ahora]. De hecho, en un artículo suyo de 1993 fueron los primeros en utilizar imágenes rotadas y reflejadas para aumentar el conjunto de datos de entrenamiento y mejorar, de ese modo, la capacidad de generalización de la red neuronal,⁶¹² una técnica que ya comentamos al estudiar el entrenamiento de redes neuronales. Este equipo de investigadores de Georgetown utilizó redes convolutivas para analizar imágenes de

⁶⁰⁷ Yann LeCun. Generalization and network design strategies. Technical Report CRG-TR-89-4, Department of Computer Science, University of Toronto, 1989b

⁶⁰⁸ Yann LeCun. Generalization and network design strategies. En R. Pfeifer, Z. Schreter, F. Fogelman, y L. Steels, editores, *Connectionism in perspective*. Elsevier, 1989a. ISBN 0444880615

⁶⁰⁹ Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, R. E. Howard, Wayne E. Hubbard, y Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. En D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 396–404. Morgan-Kaufmann, 1990a. URL <https://goo.gl/RKxWaW>

⁶¹⁰ Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, R.E. Howard, Wayne E. Hubbard, y Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989a. ISSN 0899-7667. DOI: 10.1162/neco.1989.1.4.541

⁶¹¹ Yann LeCun, Leon Bottou, Yoshua Bengio, y Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998a. ISSN 0018-9219. DOI: 10.1109/5.726791

⁶¹² Shih-Chung B. Lo, Jyh-Shyan Lin, Matthew T. Freedman, y Seong Ki Mun. Computer-Assisted Diagnosis of Lung Nodule Detection using Artificial Convolution Neural Network. En *Proceedings of SPIE, Medical Imaging VII: Image Processing*, volume 1898, pages 859–869, 1993. DOI: 10.1117/12.154572

tipo médico,⁶¹³ desarrollando sistemas automatizados para la detección de nódulos/cáncer en pulmones⁶¹⁴ y masas en mamografías.⁶¹⁵ De su trabajo también se derivaron sistemas aprobados por la FDA y utilizados en el mundo real para facilitar el diagnóstico del cáncer. Por ejemplo, al procesar una imagen médica, el sistema puede eliminar la estructura vascular de las imágenes de tomografía computerizada o las sombras de las costillas en las radiografías, lo que facilita el diagnóstico médico de enfermedades pulmonares.

Las redes convolutivas resultaron ser especialmente adecuadas para aprovechar la estructura bidimensional de los píxeles de una imagen. En las imágenes, suelen ser importantes las relaciones de adyacencia entre píxeles y las redes convolutivas explotan este hecho para obtener resultados que no se podrían conseguir con una red multicapa tradicional. Pero no sólo son útiles para imágenes, sino que también se han empleado con éxito en aplicaciones que trabajan sobre otros tipos de señales, como los sistemas de reconocimiento de voz.^{616,617}

De hecho, exactamente el mismo algoritmo que sobre imágenes emplea convoluciones en dos dimensiones y dio lugar a las redes convolutivas modernas, se puede aplicar sobre señales unidimensionales realizando las convoluciones en una única dimensión, usualmente la dimensión tiempo. Son las redes TDNN [*Time-delay neural networks*], que podemos ver como redes convolutivas 1D sobre series temporales.⁶¹⁸ El objetivo de una red TDNN es clasificar patrones que se pueden desplazar en el tiempo [*shift-invariant*], de forma que no se requiera establecer de antemano dónde empieza y dónde termina el patrón. Las redes TDNN se propusieron inicialmente para clasificar fonemas en señales de voz para construir sistemas de reconocimiento de voz, en los que es difícil, si no imposible, determinar de forma precisa los límites de los segmentos que corresponden a cada fonema. Mediante una red TDNN, se consiguen reconocer los fonemas y sus características acústicas o fonéticas independientemente de su desplazamiento en el tiempo (de su posición exacta en la señal de voz). Las redes TDNN son redes multicapa en las que cada capa está compuesta de un conjunto de clusters, cada uno de los cuales se centra sólo en una pequeña región de la señal de entrada, por lo que la red recibe como entrada una serie de muestras consecutivas de la señal de voz. En otras palabras, las diferentes entradas de la red corresponden a distintos retardos introducidos sobre la señal de entrada de la red, de donde proviene el nombre de la red TDNN. De esta forma, se consigue añadir una componente temporal a la red sin necesidad de incluir conexiones sinápticas de realimentación como las utilizadas por las redes recurrentes y la red se puede entrenar como una red multicapa convencional.

Una vez analizados los antecedentes históricos de las redes convolutivas, es el momento de definir exactamente en qué consiste la operación matemática de convolución y diseñar los distintos módulos con los que

⁶¹³ Shih-Chung B. Lo, Heang-Ping Chan, Jyh-Shyan Lin, Huai Li, Matthew T. Freedman, y Seong K. Mun. Artificial Convolution Neural Network for Medical Image Pattern Recognition. *Neural Networks*, 8(7/8):1201 – 1214, 1995a. ISSN 0893-6080. DOI: 10.1016/0893-6080(95)00061-5

⁶¹⁴ Shih-Chung B. Lo, S.L.A. Lou, Jyh-Shyan Lin, Matthew T. Freedman, M.V. Chien, y Seong Ki Mun. Artificial Convolution Neural Network Techniques and Application for Lung Nodule Detection. *IEEE Transactions on Medical Imaging*, 14(4):711–718, 1995b. ISSN 0278-0062. DOI: 10.1109/42.476112

⁶¹⁵ Shih-Chung B. Lo, Huai Li, Yue Wang, Lisa Kinnard, y Matthew T. Freedman. A Multiple Circular Path Convolution Neural Network System for Detection of Mammographic Masses. *IEEE Transactions on Medical Imaging*, 21(2):150–158, 2002. ISSN 0278-0062. DOI: 10.1109/42.993133

⁶¹⁶ Ossama Abdel-Hamid, Abdel rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, y Dong Yu. Convolutional Neural Networks for Speech Recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(10):1533–1545, 2014. ISSN 2329-9290. DOI: 10.1109/TASLP.2014.2339736

⁶¹⁷ Ossama Abdel-Hamid, Li Deng, y Dong Yu. Exploring Convolutional Neural Network Structures and Optimization Techniques for Speech Recognition. En *INTERSPEECH 2013, 14th Annual Conference of the International Speech Communication Association, Lyon, France, August 25-29, 2013*, pages 3366–3370. ISCA, 2013. URL <https://goo.gl/1axXVc>

⁶¹⁸ Kevin J. Lang y Geoffrey E. Hinton. The Development of the Time-Delay Neural Network Architecture for Speech Recognition. Technical Report CMU-CS-88-152, Department of Computer Science, Carnegie-Mellon University, 1988

Las redes recurrentes también son capaces de manejar datos temporales, si bien lo hacen de forma completamente distinta. En lugar de recibir simultáneamente entradas correspondientes a distintos instantes de tiempo, las redes recurrentes mantienen su estado para recordar sus entradas pasadas (y, en el caso de las redes recurrentes bidireccionales, también las futuras). Esto es, las redes recurrentes tienen memoria, las redes TDNN no la tienen.

podremos construir redes convolutivas capaces de trabajar con diferentes tipos de señales.

La operación de convolución

Las redes convolutivas son, simplemente, redes multicapa en las que algunas capas realizan una operación de convolución en lugar de la tradicional multiplicación matricial de entradas por pesos.

Matemáticamente, la convolución es una operación matemática que se realiza sobre dos funciones para producir una tercera que se suele interpretar como una versión modificada (filtrada) de una de las funciones originales. La convolución de las funciones f y g , que se suele denotar mediante un asterisco $*$ o una estrella \star , se define como la integral del producto de dos funciones después de que una de ellas se refleje y se desplace:

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

En procesamiento digital de señales, cuando utilizamos señales discretas, la integral anterior se convierte en una sumatoria:

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

Normalmente, uno de los operandos de la convolución es la señal que deseamos procesar, $x[n]$, y el otro corresponde al filtro, $h[n]$, con el que procesamos la señal. Cuando el filtro es finito y se define sólo sobre el dominio $\{0, 1, \dots, K - 1\}$, la operación de convolución consiste en, para cada valor de la señal, realizar K multiplicaciones y $K - 1$ sumas:

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

La operación de convolución, que hasta ahora hemos definido sobre funciones de una variable, se puede extender fácilmente al caso multidimensional. En el caso de señales discretas definidas sobre dos variables a las que se aplica un filtro de tamaño $K_1 \times K_2$, la convolución se calcula utilizando la siguiente expresión:

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

En procesamiento digital de imágenes, en las que las variables $[n_1, n_2]$ corresponden a las coordenadas $[x, y]$ de los píxeles de una imagen, el signo menos que aparece en la definición de la operación de convolución

En aplicaciones de ingeniería, resulta habitual utilizar la notación $f[n] \star g[n]$ en lugar de $(f \star g)[n]$.

se suele sustituir por un signo más, de forma que la convolución se calcula como:

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

Técnicamente, no se trata de una convolución, sino de una operación similar denominada correlación cruzada. En el caso discreto para señales reales, la única diferencia es que el filtro utilizado $h[x, y]$ aparece reflejado con respecto a la definición formal de convolución. Dado que la diferencia es mínima, en muchas ocasiones se habla de convolución cuando, realmente, se está calculando una correlación cruzada.

Para comprender mejor en qué consiste la operación de convolución, utilicemos un ejemplo sencillo. Supongamos que tenemos una imagen en blanco y negro captada por una cámara. Dicha imagen, de 7×7 píxeles, se puede representar mediante una matriz binaria:

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

Centrado en la imagen aparece un pequeño objeto cuadrado. Veamos qué sucede cuando, a la imagen anterior, le aplicamos un filtro que definimos utilizando la siguiente máscara:

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

Sólo tenemos que ir situando dicho filtro sobre diferentes posiciones de la imagen para obtener la imagen filtrada. Por ejemplo, si situamos la máscara del filtro centrado sobre el píxel que corresponde a la esquina superior izquierda del objeto representado en la imagen, sólo tenemos que fijarnos en los píxeles alrededor de esa posición para calcular el resultado de la convolución sobre ese pixel:

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

Sólo tenemos que ir multiplicando elemento a elemento y sumando el resultado: $0 * 0 - 1 * 0 + 0 * 0 - 1 * 0 + 4 * 1 - 1 * 4 + 0 * 0 - 1 * 1 + 0 * 1 = 4 - 1 - 1 = 2$. En ese cálculo, para el píxel de la esquina superior izquierda del objeto, todos los demás píxeles de la imagen son irrelevantes.

Si repetimos el proceso para las todas las posiciones en las que se puede situar la máscara sobre la imagen, obtenemos el resultado de la convolución:

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

Mediante la convolución hemos conseguido identificar los píxeles de la imagen que corresponden a las esquinas y a las fronteras del objeto que aparecía en la imagen original.

Observe que, tal como hemos realizado la convolución, la imagen obtenida como resultado es de tamaño 5×5 , más pequeña que la imagen original de 7×7 . Si, por algún motivo, deseamos obtener una imagen del mismo tamaño que la original, una suposición habitual es asumir que los píxeles de la imagen están rodeados por ceros [*zero padding*], tantos como nos hagan falta para poder obtener una imagen de las mismas dimensiones que la original. En nuestro caso, la imagen resultante sería de la forma:

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

La operación de convolución resulta particularmente interesante en procesamiento digital de señales porque permite construir la salida de un sistema para cualquier señal de entrada arbitraria si conocemos la respuesta impulsiva del sistema (la salida del sistema cuando se introduce un impulso como entrada).

En procesamiento de imágenes, muchas de las operaciones utilizadas en programas de retoque fotográfico se pueden representar de forma compacta definiendo la máscara o kernel del filtro que se aplica sobre la imagen realizando una convolución:

- *Detección de fronteras [edge detection]*

El ejemplo que utilizamos para ilustrar la operación de convolución es un detector de fronteras sencillo, del que podemos diseñar variantes

El resultado de la convolución con *zero padding* dependerá de las propiedades de la imagen y del filtro utilizado. No siempre tendrá un borde vacío como en el ejemplo.

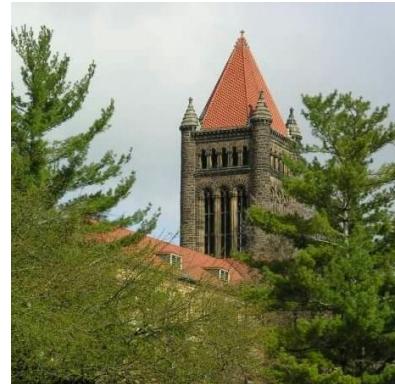


Figura 149: La imagen utilizada para ilustrar las operaciones de procesamiento de imágenes que se pueden realizar con una simple convolución: Altgeld Hall, sede del Departamento de Matemáticas de la Universidad de Illinois en Urbana-Champaign.

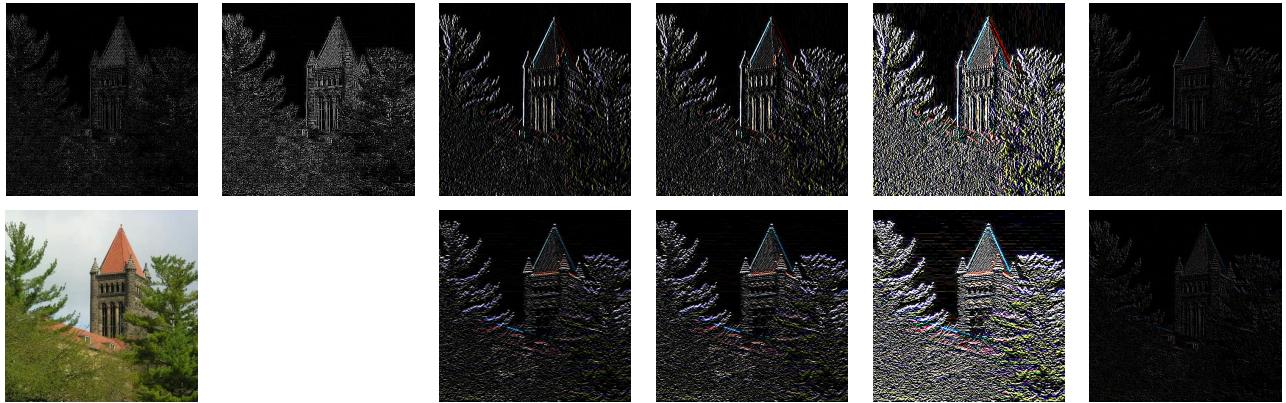


Figura 150: Detección de fronteras: Los detectores de fronteras H_4 , H_8 , P , SF , S y R . Para los operadores que distinguen entre fronteras horizontales y verticales, la fila superior muestra la detección de fronteras horizontales y la fila inferior reproduce las fronteras verticales detectadas.

con facilidad en función de cómo queramos resaltar los cambios entre píxeles adyacentes de la imagen.

La versión anterior detecta, simultáneamente, fronteras horizontales y verticales:

$$H_4 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

La siguiente versión podemos utilizarla para detectar, además, fronteras en diagonal:

$$H_8 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

El operador de Prewitt fue propuesto por Judith M. S. Prewitt, de la Universidad de Pennsylvania, como un operador de diferenciación discreta que calcula una aproximación del gradiente de la intensidad de la imagen, horizontal o verticalmente:

$$P_x = \begin{bmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} +1 & 0 & -1 \end{bmatrix}$$

$$P_y = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} +1 \\ 0 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

El filtro de Sobel, también conocido como operador de Sobel-Feldman, fue propuesto en 1968 por Irwin Sobel y David Feldman, del Laboratorio de IA de la Universidad de Stanford, como un gradiente suavizado que le da más peso al píxel central que a los adyacentes:

$$SF_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} +1 & 0 & -1 \end{bmatrix}$$

$$SF_y = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} +1 \\ 0 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

El grado de suavizado es ajustable, por lo que también podríamos utilizar operadores similares con las propiedades que nos interesen. Por ejemplo, Hanno Scharr propuso:

$$S_x = \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix} = \begin{bmatrix} 3 \\ 10 \\ 3 \end{bmatrix} \begin{bmatrix} +1 & 0 & -1 \end{bmatrix}$$

$$S_y = \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} = \begin{bmatrix} +1 \\ 0 \\ -1 \end{bmatrix} \begin{bmatrix} 3 & 10 & 3 \end{bmatrix}$$

en los que la máscara posee la siguiente propiedad $[3\ 10\ 3] = [3\ 1] * [1\ 3]$.

No es necesario que el filtro utilizado para detectar fronteras sea de tamaño 3×3 . Se pueden definir operadores de Prewitt, Sobel-Feldman y Scharr de tamaño 5×5 o 7×7 , que pueden resultar útiles en la práctica para imágenes con un número elevado de píxeles.

El operador de Roberts fue uno de los primeros detectores de fronteras, propuesto por Lawrence Roberts, del Laboratorio Lincoln del MIT, en 1963. Tampoco utilizaba máscaras de tamaño 3×3 , sino máscaras de tamaño 2×2 , por lo que resulta más eficiente computacionalmente que los demás operadores mencionados. La idea del detector de fronteras de Roberts consistía en sumar los cuadrados de las diferencias entre píxeles adyacentes diagonalmente. Su objetivo era conseguir aristas bien definidas, en las que el fondo aportase poco ruido y la intensidad de las aristas correspondiese a lo que percibimos los humanos, de acuerdo a las teorías psicofísicas de la época. En el detector de Roberts, hay que realizar una convolución con las siguientes dos máscaras:

$$R_+ = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

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

■ *Suavizado de imágenes [blur/smoothing]*

Si nos fijamos, todas las máscaras utilizadas para detectar fronteras calculan diferencias entre los valores de píxeles cercanos, lo que les permite resaltar las diferencias que existen entre ellos. Si, en lugar de calcular diferencias para aproximar un gradiente, promediamos los píxeles adyacentes, el resultado de la convolución será una versión suavizada de la imagen original. Esta versión suavizada, con menos

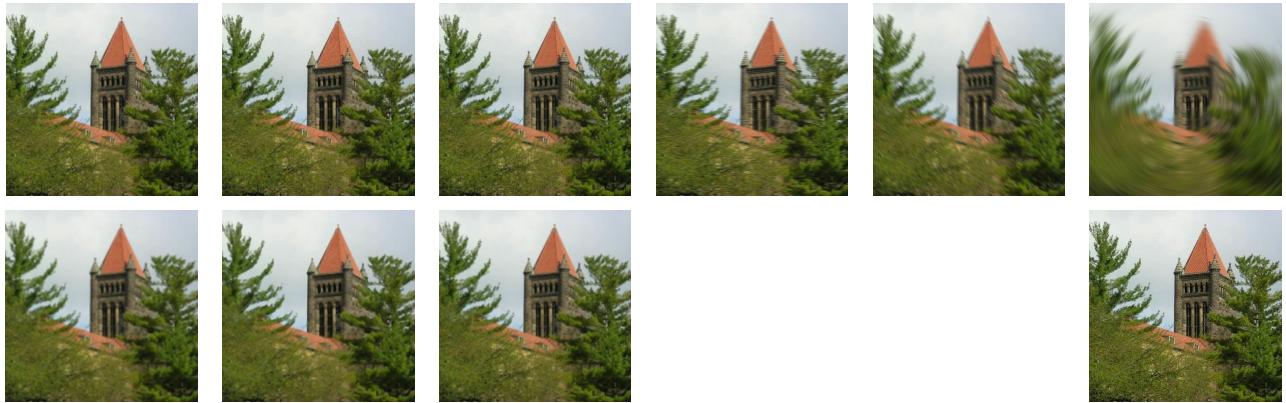


Figura 151: Desenfoque/suavizado de imágenes: A la izquierda, filtro de media, de mediana y gaussiano, con máscaras de tamaño 3×3 en la fila superior y de tamaño 5×5 en la fila inferior. A la derecha, desenfoque de movimiento: horizontal, diagonal y radial, este último de tipo artístico. Abajo a la derecha, la imagen original.

ruido que la original, también difuminará las fronteras de la imagen, por lo que se verá algo más borrosa que la imagen de partida.

Lo más sencillo es calcular la media aritmética de un conjunto de píxeles adyacentes utilizando máscaras de tamaño 3×3 o 5×5 :

$$B_{3 \times 3} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$B_{5 \times 5} = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Un efecto similar se consigue con un filtro de mediana, en el que la intensidad de cada pixel de la imagen final es la mediana de los píxeles adyacentes, si bien este filtro no puede representarse en forma de convolución de imágenes.

Se puede conseguir un efecto más natural de suavizado si, en lugar de realizar una media aritmética, se aplica un suavizado de tipo gaussiano [*gaussian blur*], con máscaras como las siguientes:

$$G_{3 \times 3} = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$G_{5 \times 5} = \frac{1}{52} \begin{bmatrix} 1 & 1 & 2 & 1 & 1 \\ 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \\ 1 & 1 & 2 & 1 & 1 \end{bmatrix}$$



Los ejemplos anteriores suavizan la imagen de forma isotrópica (de la misma manera en todas direcciones). También se puede conseguir un efecto vistoso si hacemos que el suavizado se efectúe en una dirección particular, lo que da lugar a una imagen desenfocada similar a la que se obtendría si se mueve la cámara al tomar una fotografía. Es el desenfoque de movimiento [*motion blur*]:

$$M_{horizontal} = \frac{1}{5} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$M_{diagonal} = \frac{1}{5} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

■ *Afilado de imágenes [sharpening]*

El principal inconveniente de los filtros de suavizado de imágenes es que, junto al ruido que puede aparecer en píxeles individuales de la imagen, también se difuminan las fronteras de los objetos representados en la imagen.

Para perfilar mejor una imagen no demasiado nítida, podemos utilizar un detector de fronteras modificado como los siguientes:

$$S_5 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$S_9 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Las máscaras anteriores no son más que versiones del primer detector de fronteras que vimos. Si nos fijamos, mientras que los pesos que aparecen en las máscaras de los detectores de fronteras suman 0, aquí

Figura 152: Afilado de imágenes: De izquierda a derecha, imagen original, filtro S_5 , filtro S_9 , filtro $S_{5 \times 5}$ y afilado conseguido con un programa de retoque fotográfico que utiliza un filtro de paso alto para procesar la imagen resaltando fronteras (mostradas en la última imagen de la serie).

hemos utilizado máscaras que suman 1, añadiendo al resultado del detector de fronteras el valor del píxel central.

Los resultados conseguidos con filtros como los anteriores no son demasiado buenos, por lo que podemos intentar combinar en una única máscara un detector de fronteras y un filtro de suavizado. Por ejemplo:

$$S_{5 \times 5} = \frac{1}{5} \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & 2 & 8 & 2 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

Los programas de retoque fotográfico utilizan este tipo de técnicas para mejorar de forma automática el aspecto visual de una imagen, si bien recurren a técnicas algo más sofisticadas para conseguir un efecto más natural en el resultado.

- *Grabados [embossing]*

Utilizando una estrategia similar, se pueden conseguir fácilmente efectos muy llamativos, como la sensación de que la imagen está tallada en relieve. Para ello, basta con definir una máscara como la siguiente:

$$E = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

No todas las operaciones que se pueden realizar sobre imágenes se pueden especificar a través de una sencilla máscara de convolución. El filtro de mediana que mencionamos como posible técnica de suavizado de imágenes es una de ellas. Las operaciones de procesamiento de imágenes morfológico, como la erosión o la dilatación de imágenes, están basadas en teoría de conjuntos y no se pueden representar como una convolución simple. Pese a ello, los ejemplos que hemos visto dan una idea de la versatilidad que ofrece la operación de convolución con máscaras de tamaño reducido a la hora de extraer, de las imágenes, características que pueden ser de nuestro interés.

Obviamente, nuestra intención es no tener que diseñar manualmente las máscaras de convolución, sino disponer de un algoritmo automático de aprendizaje que sea capaz de descubrir cuáles son las máscaras más adecuadas para el problema que nos ocupe. Como la convolución es una operación lineal, que se puede interpretar como un simple producto escalar de dos vectores, resulta sencillo diseñar redes neuronales en las que se utilicen convoluciones cuyos parámetros se aprendan con gradiente descendente y *backpropagation*. Al fin y al cabo, calcular la derivada del error cuando se trata de una convolución no difiere tanto de calcular la

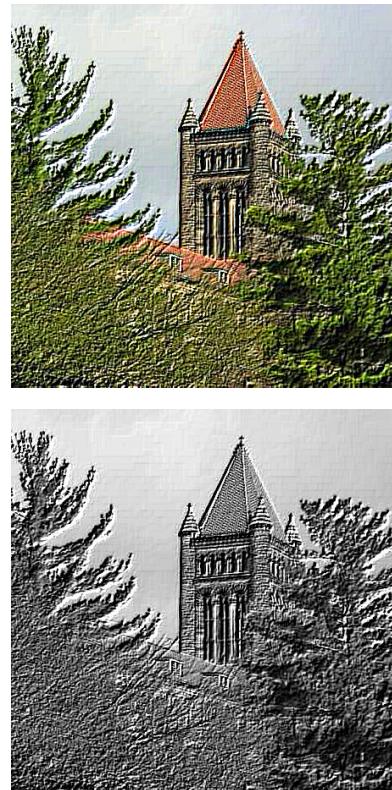


Figura 153: Un grabado de la imagen original, en color y en blanco y negro.

derivada del error con respecto a los pesos o las entradas de una neurona tradicional, los cuales se combinan usando un producto escalar para obtener la entrada neta de la neurona.

Capas convolutivas

El componente clave de las redes convolutivas son las capas en las que una convolución sustituye a la tradicional multiplicación de pesos por entradas. En el fondo, una capa convolutiva no hace más que recurrir a técnicas *ad hoc* que nos permiten incorporar conocimiento acerca del problema en el diseño de una red neuronal. En este caso, ese conocimiento proviene de las características típicas de las señales a las que se aplican las redes convolutivas: invarianza en el tiempo para señales temporales (p.ej. audio), invarianza en el espacio para señales espaciales (p.ej. imágenes) o ambas para señales espacio-temporales (p.ej. vídeo).

La entrada de una capa convolutiva es una señal, unidimensional en el caso de sonidos, bidimensional en el caso de imágenes. Dicha señal se procesa realizando una convolución con una máscara, también llamada *kernel*. En lugar de diseñar ese kernel manualmente, sus valores se aprenden. Por tanto, los pesos correspondientes a la máscara de convolución serán los parámetros de la capa convolutiva que hemos de entrenar.

A diferencia de lo que sucede en las capas tradicionales de una red neuronal multicapa, no todas las entradas están conectadas con todas las salidas. Cada neurona individual de una capa convolutiva opera sobre un subconjunto de las entradas (un fragmento de la señal de voz o una región de la imagen). Por tanto, cada neurona que se emplea detecta una característica local de la señal de entrada.

En el caso de las imágenes, cada neurona detecta una característica en una región diferente de la imagen. Ahora bien, si una de las neuronas consigue detectar una característica útil en una región particular de la imagen, tiene sentido que se aproveche esa característica aprendida para que la red sea capaz de detectar la misma característica en otras regiones de la imagen. Por este motivo, se utilizan detectores replicados para que la red sea capaz de detectar una característica particular en cualquier posición de la imagen. Es decir, se utilizan los mismos pesos (la misma máscara de convolución) en neuronas diferentes cuyas entradas están conectadas a diferentes regiones de la imagen.

La existencia de detectores replicados no hacen que la actividad neuronal sea invariante frente a translaciones, sino equivariante. Si desplazamos la señal de entrada, la salida de la capa convolutiva no será la misma (invarianza), sino que aparecerá también desplazada en el mismo sentido que la entrada (equivarianza). Matemáticamente, una función es equivariante si, cuando la entrada cambia, la salida cambia de la misma forma:

$f(g(x)) = g(f(x))$. Las capas convolutivas son, por tanto, equivariantes frente a traslaciones.

En resumen, una capa convolutiva se caracteriza por incorporar en su arquitectura dos características clave:

- Uso de conexiones locales que restringen la conectividad de la red: Las regiones de la señal de entrada a las que están conectadas las diferentes neuronas se suelen denominar campos receptivos [locales] por su analogía con los campos receptivos de las neuronas de la retina y del córtex visual.
- Uso de pesos compartidos por varias neuronas [*weight sharing*]: El uso de detectores replicados reduce el número de parámetros (pesos) que deben ajustarse, lo que facilita el entrenamiento de la red neuronal.

Dado que se requiere un conjunto completo de neuronas para detectar una única característica en distintas posiciones de la entrada, no tiene demasiado sentido que una capa convolutiva sólo sea capaz de detectar una característica particular. En la práctica, en una capa convolutiva se incluyen distintos tipos de detectores de características, todos ellos replicados para detectar las distintas características en diferentes posiciones de la entrada.

Cada tipo de detector corresponde a una máscara de convolución concreta, un conjunto de pesos que han de ajustarse. Como la salida de cada tipo de detector es, a su vez, una señal del mismo tipo que la señal de entrada (p.ej. una imagen en la que se muestra en qué posiciones de la imagen de entrada aparece la característica detectada), se suele hablar de mapas de características [*feature maps*]. Cada tipo de detector generará un mapa diferente, por lo que la salida de una capa convolutiva capaz de detectar K características diferentes estará formada por K mapas de características. Es decir, dada una imagen de entrada, la salida de la capa convolutiva estará formada por K imágenes distintas en las que cada fragmento de la imagen original se representa de K maneras diferentes.

Hiperparámetros de una capa convolutiva

Al diseñar una capa convolutiva, tenemos que establecer valores para cuatro hiperparámetros. Los dos primeros ya nos los hemos encontrado: el número de filtros K y su esquema de conectividad local (i.e. el tamaño de sus máscaras de convolución). Los dos siguientes se emplean por cuestiones computacionales de índole práctico: el uso de relleno con ceros [*zero padding*] y su paso, salto o zancada [*stride*].

- *Profundidad*

El primer hiperparámetro de una capa convolutiva es el número de filtros que incluye, cada uno de ellos capaz de aprender a detectar una

característica diferente en la entrada de la capa. Por ejemplo, si la entrada es una imagen, las neuronas correspondientes a distintos mapas de características se activarán ante la presencia de diferentes patrones en la imagen de entrada, como aristas con diferentes orientaciones (horizontales, verticales, inclinadas) o segmentos de un color particular.

El número de filtros K de una capa convolutiva determina su profundidad, el número de características que es capaz de detectar. El conjunto de neuronas que detecta las K características en una región particular de la entrada se suele denominar columna, aunque hay quien prefiere llamarlas fibra [*fibre*].

- *Conejividad local*

Cuando se trabaja con señales, no es práctico que todas las neuronas estén conectadas a todas las entradas. Cada neurona se conecta sólo a una región local de la entrada de la capa. La extensión temporal o espacial de esa región es un hiperparámetro F al que se denomina tamaño del filtro o del campo receptivo de la neurona.

Al trabajar con imágenes, aunque no sea estrictamente obligatorio, es habitual que la extensión espacial del campo receptivo local de cada neurona sea la misma en horizontal y en vertical. Normalmente, se emplean filtros de convolución de tamaño reducido, p.ej. 3×3 ($F = 3$), 5×5 ($F = 5$) o 7×7 ($F = 7$).

En ocasiones, también se utilizan convoluciones 1×1 ($F = 1$). Aunque pueda parecer contra intuitivo, tiene su sentido. Si partimos de una señal particular, una convolución 1×1 se limita a escalar la señal de entrada, algo que no parece demasiado útil. Sin embargo, tengamos en cuenta que las capas convolutivas también tienen profundidad. Entonces la convolución 1×1 nos permite reducir la dimensionalidad de los datos. Por ejemplo, si tenemos una imagen en color, en realidad tenemos 3 imágenes paralelas, las correspondientes a los canales R (rojo), G (verde) y B (azul). A partir de esa imagen en color, podemos obtener una imagen en escala de grises definida por su luminancia $Y = 0.177R + 0.813G + 0.011B$ de acuerdo al estándar CIE o $Y = 0.299R + 0.587G + 0.114B$ de acuerdo al estándar NTSC. Ambos son ejemplos de convolución 1×1 aplicada para combinar diferentes canales de una imagen. En redes convolutivas, el uso de convoluciones 1×1 está biológicamente inspirado:⁶¹⁹ en el córtex visual tenemos campos receptivos locales (kernels) especializados en la detección de estímulos con diferentes orientaciones espaciales, que luego combinamos sin problemas.

Obviamente, las primeras capas convolutivas de una red siempre deben tener una máscara de convolución de tamaño mayor que 1×1 , para que su campo receptivo sea capaz de capturar información espacial

⁶¹⁹ Min Lin, Qiang Chen, y Shuicheng Yan. Network in network. *arXiv e-prints*, arXiv:1312.4400, 2013. URL <http://arxiv.org/abs/1312.4400>

local. Posteriormente, las convoluciones 1×1 nos permiten aprender de las interacciones entre los distintos canales de la señal de entrada.

- *Paso, salto o zancada [stride]*

El coste computacional de realizar una convolución se puede reducir a costa de no extraer características con el mismo grado de detalle con el que aparecen en la señal de entrada. Una convolución submuestreada [*downsampled convolution*] se realiza saltándose posiciones, lo que nos permite reducir el número de operaciones aritméticas necesarias como el tamaño de la salida obtenida.

Al diseñar una capa convolutiva, podemos establecer un tamaño S de paso, salto o zancada [*stride*] que indique cada cuántas muestras de la entrada queremos obtener una muestra de salida. Cuando $S = 1$, realizamos la convolución normalmente. Cuando $S = 2$, nos saltamos una de cada dos muestras. En el caso de señales multidimensionales, podemos incluso establecer tamaños de salto diferentes para cada dimensión.

Por ejemplo, al trabajar con imágenes, $S = (1, 1)$ realizaría una convolución tradicional, píxel a píxel, y $S = (2, 2)$ realizaría una convolución submuestreada que generaría una salida 4 veces más pequeña que la entrada original. Si, por algún motivo, queremos obtener más resolución a la hora de detectar características horizontalmente, podríamos utilizar $S = (1, 2)$ para submuestrear sólo en vertical y reducir a la mitad las dimensiones de la salida de la capa convolutiva.

- *Rellenado con ceros [zero padding]*

Como vimos al describir la operación de convolución, en ocasiones resulta conveniente llenar la entrada con ceros alrededor para obtener una salida del tamaño deseado. La cantidad de relleno Z es un hiperparámetro más de una capa convolutiva. Cuando se utiliza, lo normal es emplearlo para preservar las dimensiones de la entrada de la capa. En el caso de imágenes, para conseguir que tanto la imágenes de entrada como las imágenes de salida tengan exactamente el mismo número de píxeles.

Por ejemplo, si partimos de imágenes en alta resolución, de 1920×1080 píxeles, y empleamos máscaras de tamaño 5×5 , los mapas de características de la capa convolutivas tendrán una resolución de 1918×1078 píxeles. Si queremos que la salida sea del mismo tamaño que la entrada, hemos de añadir *zero padding* con $Z = 2$. En general, si no se submuestra la convolución, tendremos que utilizar $Z = (F - 1)/2$ para que la salida de la capa convolutiva preserve las dimensiones de su entrada. Es habitual el uso de $F = 3$, $S = 1$ y $Z = 1$ en muchas aplicaciones.

El número de neuronas de una capa convolutiva es una función del tamaño de sus entradas ($W \times H$ en el caso de imágenes bidimensionales),

El uso de $S \geq 3$ es poco habitual en la práctica.

su profundidad (K), el tamaño de sus filtros (F), su *stride* (S) y el uso de *zero padding* (Z): $K * ((W - F + 2Z)/S + 1) * ((H - F + 2Z)/S + 1)$. Esta expresión nos da las dimensiones de la salida de la capa convolutiva.

Por ejemplo, al procesar una imagen de no demasiada resolución, 256×256 píxeles, usando $K = 10$ filtros de tamaño 3×3 , obtendremos una salida de tamaño $10 \times 254 \times 254$ si no utilizamos *zero padding* y de tamaño $10 \times 256 \times 256$ cuando $Z = 1$. Si queremos reducir el número de píxeles de salida (640K en 10 canales de 64K), podemos submuestrear usando $S = 2$, de forma que la salida de la capa convolutiva sea de tamaño $10 \times 128 \times 128$ (160K en 10 canales de 16K).

El ejemplo anterior nos da una idea del coste computacional que supone utilizar capas convolutivas en una red neuronal. Para procesar imágenes en alta definición, de 1920×1080 , con más de dos millones de píxeles, tenemos que realizar tantas convoluciones como características deseemos extraer. Como el coste de calcular cada píxel de salida es cuadrático con respecto al tamaño de los filtros F y proporcional tanto al tamaño de la entrada $W \times H$ como al número de filtros K , el coste computacional de emplear una capa convolutiva es $O(KWHF^2)$. El uso de submuestreo fijando un tamaño de salto $S > 1$ nos permite aligerar parcialmente este coste computacional, origen también del otro componente habitual en las redes convolutivas actuales: las capas de submuestreo o *pooling*.

En cuanto al número de parámetros de la capa convolutiva, si la capa recibe como entrada C señales (p.ej. tres imágenes correspondientes a los tres canales de una imagen en color), cada filtro convolutivo tendrá $C * F^D$ parámetros, siendo F el tamaño del campo receptivo local y D la dimensionalidad de las señales de entrada ($D = 2$ para imágenes). Como la capa convolutiva se emplea para generar K mapas de características, el número total de parámetros de la capa será $K * C * F^D$.

Compare este valor con el que obtendríamos si utilizásemos una capa convencional, sin compartir pesos. Entonces, dada una entrada de tamaño $C \times W \times H$ necesitaríamos $C * W * H$ pesos para cada neurona de salida, de las que tenemos del orden de $K * W * H$. El número de parámetros se dispararía hasta $K * C * W^2 * H^2$. Para poner cifras concretas, dada una imagen HD en color ($C = 3$, $W = 1920$, $H = 1080$), entrenar una capa convolutiva supone ajustar un número de parámetros proporcional al tamaño de los filtros (p.ej. 5×5), mientras que una capa convencional necesitaría un número de parámetros cuadrático con respecto a las dimensiones de la imagen (4×10^{12} para una imagen HD). Algo completamente inviable en la práctica. Si utilizásemos campos receptivos locales pero no compartiésemos pesos, el número de parámetros necesario seguiría siendo proporcional a las dimensiones de la entrada (del orden de 50 millones de pesos por cada mapa de características correspondiente a un filtro de tamaño 5×5). Entrenar una red neuronal con tantos parámetros no es del todo inviable, siempre que dispongamos de conjuntos de entrenamiento

enormes, pero sí sigue resultando muy poco práctico. Mejor combinar las dos características clave de las capas convolutivas, conexiones locales y pesos compartidos, para ser capaces de extraer características con sólo ajustar un puñado de parámetros (sólo 25 por cada filtro de tamaño 5×5 para cada canal de la señal de entrada, independientemente de las dimensiones de la imagen de entrada).

En definitiva, reemplazar una capa completamente conectada, en la que todas las entradas están conectadas a todas las salidas, por una capa convolutiva nos permite reducir significativamente el número de parámetros de la red que hemos de ajustar. A cambio, hemos de establecer una serie de hiperparámetros adicionales que definen la topología de la red convolutiva.

Capa convolutiva vs. capa completamente conectada

Aunque sustituimos la multiplicación de matrices de una capa completamente conectada por una convolución, las únicas diferencias que existe entre las capas convolutivas y las capas completamente conectadas es el hecho de que las neuronas de una capa convolutiva sólo se conectan a una región local de la entrada y muchas de las neuronas de una capa convolutiva comparten sus pesos. Por lo demás, matemáticamente estamos realizando la misma operación para calcular cada salida individual: un producto escalar de un vector de entradas por un vector de pesos.

De hecho, existe una correspondencia directa entre las capas convolutivas y las capas completamente conectadas:

- Dada una capa convolutiva, se puede construir una capa completamente conectada que implementa exactamente la misma función. Su matriz de pesos sería enorme, pero en su mayor parte estaría rellena de ceros. Sólo sería distinta de cero para las conexiones correspondientes a las conexiones locales de cada neurona (conectividad local) y, además, los conjuntos de pesos correspondientes a regiones locales se repetirían en muchos bloques de la matriz (pesos compartidos).
- Dada una capa completamente conectada, es posible diseñar una capa convolutiva equivalente. De manera trivial puede verse que sólo necesitamos utilizar un filtro de convolución de tamaño igual a la entrada de la capa. Sin utilizar *zero padding*, la salida de la capa convolutiva será una única columna de profundidad igual al número de entradas de la capa, idéntica a la salida de la red completamente conectada.

La segunda conversión es útil en la práctica cuando, como es habitual, en una red neuronal combinamos capas convolutivas y capas completamente conectadas. Usualmente, las primeras capas de la red serán capas convolutivas encargadas de extraer características de la entrada.

A continuación, una vez identificadas las características más útiles, se utiliza una red multicapa convencional para resolver el problema para el que esté diseñado la red (p.ej. un problema de clasificación utilizando una capa softmax como capa de salida).

La conexión entre la salida de las capas convolutivas y la entrada de las capas convencionales sólo requiere reorganizar la estructura del array de datos con el que se trabaja (un vector en el caso unidimensional, una matriz en el bidimensional o un tensor en general).

Sin embargo, interpretar las capas finales como si fueran convolutivas puede ayudarnos a conseguir una implementación eficiente de una red de este tipo cuando deseamos utilizar la red sobre diferentes regiones de una misma imagen (por ejemplo, para detectar los objetos que aparecen en ella). En lugar de aplicar la red de forma independiente sobre cada una de las regiones en las que queremos detectar objetos, podemos convertir las capas completamente conectadas en capas convolutivas y pasarle, de una vez, la imagen completa. En la salida se obtendrán, simultáneamente, los valores correspondientes a diferentes regiones de la imagen. Esto supone un ahorro cuando las regiones se solapan (al utilizar una ventana deslizante) y facilita la paralelización de la implementación cuando se utiliza una GPU.

Supongamos que tenemos una red entrenada para clasificar imágenes en color de tamaño $224 \times 224 \times 3$, como las utilizadas en competiciones de visión artificial como ImageNet. Supongamos, además, que dicha red reduce el tamaño de la entrada en un factor de 32 antes de llegar a sus capas completamente conectadas, tal como hace AlexNet. Esto es, la red convierte las entradas de tamaño 224×224 en arrays de características de tamaño 7×7 , al ser $224/32 = 7$, ante de pasar a sus capas completamente conectadas.

Nos gustaría poder utilizar esa red para identificar objetos en imágenes más grandes, digamos imágenes en HD de 1920×1088 . Una posible solución sería ir seleccionando regiones de 224×224 píxeles, parcialmente solapadas, utilizando una ventana deslizante de tal forma que seleccionamos regiones de la imagen HD cada 32 píxeles. Entonces, tendremos que evaluar la red sobre 2040 regiones diferentes de la imagen HD, ya que $1920/32 = 60$, $1088/32 = 34$ y $60 * 34 = 2040$. En lugar de aplicar la red de forma independiente 2040 veces, podemos pasarle la imagen HD a la red convolutiva de una sola vez. La salida obtenida nos proporcionará el mismo resultado y, además, será más eficiente, ya que las regiones se solapan y evitaremos realizar cálculos duplicados. Si la red que estamos utilizando clasificaba imágenes 224×224 para obtener una distribución softmax, aplicada sobre la imagen 1920×1088 obtendrá 60×34 distribuciones de clases, una para cada región de la imagen.

¿Qué sucede si queremos aplicar la red sobre una imagen utilizando una ventana deslizante con un tamaño de paso que no corresponde al factor

Hemos modificado ligeramente la resolución de una imagen HD para que sus dimensiones sean múltiplo de 32.

de reducción de la red convolutiva original? Por ejemplo, si queremos probar con ventanas deslizantes cada 16 píxeles, sólo tendremos que aplicar la red convolutiva dos veces. La primera, sobre la imagen original. La segunda, sobre la imagen desplazada 16 píxeles tanto horizontal como verticalmente. Utilizar la red dos veces sobre dos imágenes ligeramente distintas será mucho más eficiente que utilizarla 4080 veces sobre imágenes del mismo tamaño con el que se entrenó originalmente la red.

A parte de utilizar esta estrategia para detectar objetos en diferentes regiones de la imagen, también podemos aprovecharla para emplear la red original, entrenada sobre imágenes más pequeñas, para clasificar imágenes de mayor resolución. Para obtener una clasificación de la imagen de mayor resolución, sólo tenemos que promediar las distribuciones obtenidas para las diferentes regiones de la imagen original (2040 en el ejemplo anterior). O eso, o reescalamos la imagen original para que su tamaño encaje con el tamaño de las imágenes con las que se entrenó la red original, con el riesgo de perder muchos detalles de la imagen de la que disponemos.

Implementación

Una capa convolutiva no es más que una capa convencional con una conectividad restringida y un conjunto de parámetros reducido, compartido entre todas las neuronas del mismo mapa de características. Los parámetros de la capa convolutiva corresponden a un conjunto de filtros, un conjunto de máscaras de convolución que se aplican localmente sobre diferentes regiones de la entrada. Por ejemplo, en una imagen en color, con tres canales, los filtros pueden ser de tamaño $5 \times 5 \times 3$ (5×5 correspondientes a la dimensión espacial de la convolución, 3 por la profundidad de la entrada).

Veamos cómo se implementaría utilizando el diseño modular que ya vimos para otros tipos de capas, para los que hemos de definir dos métodos: uno para obtener la salida de la capa a partir de la entrada y otro para propagar el error hacia atrás cuando entrenamos la red usando el gradiente descendente y *backpropagation*.

```
class Layer(n,m):
    function y = forward(x);
    function backward(error);
```

En el caso de una capa convolutiva, supongamos que la entrada es una imagen con c canales y una resolución de $w \times h$ píxeles. Además, los m mapas de características de la capa serán el resultado de aplicar filtros de tamaño k (máscaras de dimensión $k \times k$ en el caso de imágenes bidimensionales). Es decir, la entrada de la red será de tamaño $c \times w \times h$ y su salida será de tamaño $m \times (w-k+1) \times (h-k+1)$. Para no complicar demasiado el pseudocódigo, asumimos un tamaño de paso [*stride*] $s=1$ y no utilizamos *zero padding*.

La declaración de nuestra clase `ConvolutionalLayer` quedaría como:

```
class ConvolutionalLayer(c,w,h,m,k)
:Layer(c*w*h, m*(w-k+1)*(h-k+1))
```

La capa recibe una entrada de profundidad c y genera una salida de profundidad m utilizando filtros $k \times k$. Para procesar una entrada, podemos suponer que nos llega un array tridimensional `input [c] [x] [y]` correspondiente a los c canales de la imagen, a cuyos píxeles individuales accedemos mediante las coordenadas x e y . La salida `output [f] [x] [y]` se obtiene realizando una convolución con los m kernels de tamaño $k \times k$ correspondientes a cada uno de los mapas de características de la capa:

```
class ConvolutionalLayer(c,w,h,m,k)
    function output = forward(input)
        for f=1:m
            for x=1:w-k+1
                for y=1:h-k+1
                    output [f] [x] [y] = convolve(input,kernel [f],x,y)+bias [f];
```

donde aparece un sesgo `bias [f] [x] [y]` como parámetro adicional asociado a cada mapa de características (equivalente al sesgo de las neuronas individuales) y representamos mediante un array 4D el conjunto de kernels convolutivos, de forma que `kernel [f]` hace referencia al f -ésimo filtro convolutivo de la capa.

Sólo tenemos que implementar la función auxiliar que se encarga de efectuar la convolución píxel a píxel:

```
class ConvolutionalLayer(c,w,h,m,k)
    function result = convolve(input,kernel,x,y)
        result = 0;
        for canal=1:c
            for i=1:k
                for j=1:k
                    result += kernel [canal] [i] [j]*input [canal] [x+i] [y+j];
```

La realización de la convolución que permite obtener una salida individual requiere $c * k^2$ multiplicaciones. Como la capa tiene del orden de $m * w * h$ neuronas, el coste computacional de calcular la salida de la capa convolutiva es de orden $O(m * c * w * h * k^2)$. En general, de orden $O(m * c * s^d * k^d)$, donde d es el número de dimensiones de la señal de entrada y s es el tamaño de cada dimensión de la entrada.

Dado el elevado coste computacional de la convolución, en la práctica resulta esencial paralelizar el cálculo, algo que podemos conseguir utilizando la potencia de cómputo de una GPU. No se reduce el coste computacional de la operación, que sigue siendo de orden $O(k^d)$, pero

sí el tiempo de reloj necesario para obtener un resultado. En ocasiones, podemos recurrir a algoritmos alternativos para realizar la convolución de una forma más eficiente:

- *Transformada rápida de Fourier [FFT, Fast Fourier Transform]*

La operación de convolución en el dominio temporal (o espacial, en el caso de las imágenes), se puede realizar de forma alternativa en el dominio de las frecuencias como un producto punto a punto. Para ello, se obtiene la representación en frecuencias de la señal y el kernel utilizando la transformada rápida de Fourier, se realiza el producto en el dominio de la frecuencia y se realiza la transformada inversa para volver a obtener, ya en el dominio habitual, el resultado de la convolución. Si se utiliza una implementación trivial de la transformada de Fourier no se gana nada desde el punto de vista computacional, pero si recurrimos a la transformada rápida de Fourier podemos sustituir un algoritmo cuadrático de orden $O(k^2)$ en un algoritmo de orden $O(k \log k)$, lo que puede compensar si deseamos utilizar filtros de tamaño elevado.

- *Kernels separables*

Un kernel separable es un kernel d -dimensional que se puede expresar como el producto matricial de d vectores. Si recurrimos a kernels separables, una convolución en d dimensiones se puede realizar como una serie de d convoluciones unidimensionales, lo que reduce la complejidad del algoritmo de $O(k^d)$ a $O(dk)$.

- *Convolución aproximada*

Aun cuando el kernel no sea separable, podemos obtener una aproximación del kernel que sí sea separable. Esto nos permitiría reducir la complejidad computacional del cálculo a costa de obtener un resultado aproximado.

Durante el paso hacia adelante, se desliza cada filtro de la capa convolutiva por la imagen de entrada para generar los diferentes mapas de características. Esos mapas de características son mapas de activación que indican la respuesta, en cada posición, de la señal de entrada a los filtros. En una red entrenada, cada filtro aprenderá a detectar alguna característica relevante de la entrada, lo que luego le permitirá peinar la imagen de entrada en busca de regiones de la imagen donde esa característica esté presente.

Ahora bien, para poder entrenar la red, necesitamos también implementar un método que propague la señal de error hacia atrás. El paso hacia atrás de una capa convolutiva es también una convolución, aunque hemos de ser cuidadosos a la hora de realizar esa convolución para que cada entrada reciba la parte proporcional del error que contribuye a

generar y cada parámetro su contribución al error en todas las posiciones donde se aplica el filtro. En pseudocódigo:

```
class ConvolutionalLayer(c,w,h,m,k)
    function backward(error)
        inputDelta = 0;
        kernelDelta = 0;
        for f=1:m
            backward(error, f);
```

Para cada uno de los kernels correspondientes a los filtros de convolución de la capa convolutiva, realizamos el siguiente cálculo:

```
class ConvolutionalLayer(c,w,h,m,k)
    function backward(error, f)
        sum = 0;
        for x=1:w-k+1
            for y=1:h-k+1
                delta=error[f][x][y];
                // dE/dw = x
                for s=1:k
                    for t=1:k
                        kernelDelta[f][s][t] += delta*input[f][x+s][y+t];
                // dE/dx = w
                for s=1:k
                    for t=1:k
                        inputDelta[f][x+s][y+t] += delta*kernel[f][s][t];
                sum += delta;
        biasDelta[k] = sum;
```

Como siempre, los deltas corresponden a las derivadas del error con respecto a cada uno de los parámetros de la capa convolutiva (los pesos de los filtros de convolución `kernelDelta`, así como los sesgos de los filtros, `biasDelta`) y con respecto a las entradas de la capa, `inputDelta`, para poder propagar el error hacia atrás en una red multicapa con múltiples capas convolutivas.

Durante la propagación hacia atrás del error, cada neurona de la capa calcula el gradiente del error para sus pesos y para sus entradas. Como esos mismos pesos se utilizan en todas las neuronas de un mapa de características y las mismas entradas se utilizan en todos los filtros de la capa convolutiva, esos gradientes observados localmente han de agregarse, sumando las contribuciones locales hasta obtener el gradiente del error de la capa con respecto a sus parámetros y entradas.

Cuando se trabaja con imágenes, dado el coste computacional que supone realizar tantas convoluciones en cada recorrido de la red, es

habitual recurrir al uso de una GPU, capaz de paralelizar las operaciones necesarias para el funcionamiento de una red convolutiva.

Implementación para GPU

Como en ocasiones anteriores, recurriremos a las bibliotecas CUDA proporcionadas por NVidia para implementar una capa convolutiva aprovechando la potencia de cálculo de una GPU. En el caso de una capa convolutiva, nuestra implementación tendrá el siguiente aspecto en pseudocódigo:

```
class CUDAConvolutionalLayer(c,w,h,m,k):ConvolutionalLayer(c,w,h,m,k)

    function output = forward(input)
        cudnnConvolutionForward(1,input,kernel,algorithm,0,output);
        cudnnAddTensor(1,bias,1,output);

    function backward(error)
        cudnnConvolutionBackwardBias(1,error,0,biasDelta);
        cudnnConvolutionBackwardFilter(1,input,error,algorithm,0,kernelDelta);
        cudnnConvolutionBackwardData(1,kernel,error,algorithm,0,inputDelta);
```

En la implementación anterior:

- `cudnnConvolutionForward` es una función especializada que realiza la convolución aplicando el algoritmo especificado mediante el parámetro `algorithm`. El algoritmo podemos establecerlo manualmente, p.ej. `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM` realiza la multiplicación habitual para calcular la convolución y `CUDNN_CONVOLUTION_FWD_ALGO_FFT` realiza la convolución aplicando la transformada rápida de Fourier (FFT).
- El algoritmo más rápido para realizar una convolución se puede determinar en tiempo de ejecución utilizando una de las funciones proporcionadas por NVidia: `cudnnGetConvolutionForwardAlgorithm` (`input, kernel, output, CUDNN_CONVOLUTION_FWD_PREFER_FASTEST`).
- La llamada a `cudnnAddTensor` se limita a añadir los sesgos a los resultados de la convolución: `output += bias`, utilizando la convención habitual en muchas bibliotecas de cálculo numérico, según la cual las dos constantes mágicas `a=1` y `b=1` de una función `f(a,in,b,out)` se utilizan para realizar la operación `out = a*f(in) + b*out`, convención que también aparece en la llamada a `cudnnConvolutionForward`.
- En el caso de la propagación hacia atrás del error, se utilizan tres funciones especializadas que calculan, respectivamente, los deltas con

respecto a sesgos, kernels de convolución y entradas de la capa convolutiva.

Variantes

En nuestra implementación anterior, hemos utilizado tensores 3D (arrays de tres dimensiones) para representar imágenes bidimensionales con múltiples canales, como las imágenes en color, los kernels de convolución que han de operar sobre ellas y los mapas de características que se obtienen como resultado de la convolución. Como es lógico, también nos hicieron falta tensores 3D para calcular los gradientes del error, con respecto a las entradas 3D y a los parámetros 3D de la capa convolutiva.

La implementación anterior, no obstante, va procesando las entradas una a una. Si queremos realmente aprovechar al máximo la capacidad de cálculo de una GPU, lo habitual es que utilicemos tensores de 4 dimensiones para representar las entradas y las salidas de la capa. De esta forma, podemos procesar en paralelo todas las muestras de un minilote cuando realizamos el entrenamiento de la red por lotes o aplicar la red a múltiples imágenes simultáneamente si la capacidad de cálculo de nuestra GPU lo permite. En los tensores 4D de nuestra capa convolutiva, la primera dimensión indicará la muestra sobre la que trabajamos (única en aprendizaje *online*) y la segunda indicará el canal de la imagen (los canales R, G, y B de las imágenes en color o cualquiera de los múltiples mapas de características que una capa convolutiva es capaz de generar). Las dos últimas dimensiones harán referencia a un píxel concreto perteneciente a uno de los canales de la imagen de entrada o de los mapas de características generados por una capa convolutiva.

También existe la posibilidad de modificar el patrón de conexiones utilizadas en una capa convolutiva:

- *Capas localmente conectadas*

Las capas localmente conectadas utilizan convoluciones pero sin compartir pesos, algo a lo que algunos se refieren como convoluciones no compartidas [*unshared convolutions*]. En este caso, el conjunto de parámetros de la capa convolutiva será un tensor 6D, de forma que cada mapa de características tendrá una máscara de convolución diferente para diferentes zonas de la entrada.

Algo así puede ser útil si sabemos que, en nuestro conjunto de datos, son relevantes las características locales (campos perceptivos de las neuronas convolutivas) pero no tenemos razones para pensar que el hecho de que una característica sea útil en una región de la imagen sea también útil en otras regiones diferentes de la imagen (la hipótesis en que se basa el hecho de compartir parámetros entre todas las neuronas de un mapa de características). Obviamente, esto hace que el número de parámetros de la capa convolutiva se dispare.

También podemos utilizar capas localmente conectadas para modelar interacciones entre los canales de entrada y de salida de una capa convolutiva (recuerde el uso de convoluciones 1×1). En este caso, puede que la capa localmente conectada contribuya a reducir el número total de parámetros de la red neuronal.

- *Convolución en mosaico [tiled convolution]*^{620,621}

Podemos llegar a un compromiso entre los extremos de una capa convolutiva (tensor 4D de parámetros) y una capa localmente conectada (tensor 6D de parámetros). Hay ocasiones en las que la suposición que nos hace compartir parámetros para todas las zonas de una imagen no tenga sentido. Por ejemplo, cuando la entrada a la CNN tiene una estructura conocida, como puede ser la imagen centrada de una cara en un sistema de reconocimiento facial. En tal caso, puede que no tenga sentido identificar las mismas características en todas las regiones de la imagen, pero sí en determinadas zonas (p.ej. la barbilla, los ojos, las orejas o el pelo esperamos encontrarlos sólo en determinadas posiciones).

En lugar de aprender una misma característica para todas las posiciones de la imagen o características independientes en cada posición espacial posible, podemos aprender un conjunto de kernels que modificamos conforme nos movemos por la imagen. Se usarán diferentes filtros sobre posiciones distintas de la imagen, como en una red conectada localmente. Sin embargo, el número de parámetros de la capa se incrementará proporcionalmente al número de kernel utilizados, en lugar de proporcionalmente al tamaño de los mapas de características de salida.

Como si se tratase de un mosaico, cubrimos la imagen con una serie de regiones, denominadas teselas [*tiles*]. En las capas convolutivas tradicionales, usamos una única tesela, que cubre toda la imagen. En las capas localmente conectadas, el número de teselas corresponde al número de salidas de la capa convolutiva. Cuando la convolución se utiliza en mosaico, el número de teselas tendrá un valor intermedio entre los dos extremos, mayor que uno (propio de una capa convolutiva estándar) pero menor que el número de salidas (el caso de una capa localmente conectada).

Capas de pooling

Compartir parámetros a la vez que se usan campos receptivos locales permite controlar el número de parámetros de una red convolutiva. Pese a ello, el coste computacional de procesar imágenes en una red neuronal puede resultar bastante elevado. Por este motivo, las redes convolutivas suelen incluir un segundo componente clave: las capas de *pooling* o

⁶²⁰ Karol Gregor y Yann LeCun. Emergence of complex-like cells in a temporal product network with local receptive fields. *arXiv e-prints*, arXiv:1006.0448, 2010. URL <http://arxiv.org/abs/1006.0448>

⁶²¹ Jiquan Ngiam, Zhenghao Chen, Daniel Chia, Pang W. Koh, Quoc V. Le, y Andrew Y. Ng. Tiled convolutional neural networks. En J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, y A. Culotta, editores, *NIPS'2010 Advances in Neural Information Processing Systems 23*, pages 1279–1287. Curran Associates, Inc., 2010. URL <https://goo.gl/DoQGsz>

submuestreo [*subsampling/downsampling*].

Básicamente, una capa de pooling lo que hace es reducir el tamaño de su entrada para que las capas posteriores puedan trabajar con datos de entrada reducidos. La reducción puede hacerse para reducir la dimensión espacial de los datos [*spatial pooling*] o para reducir su profundidad, el número de canales con los que hay que trabajar [*cross-channel pooling*]. La convolución 1×1 es, esencialmente, un mecanismo de *pooling* paramétrico que combina varios canales usando los pesos del kernel de convolución.

La versión tradicional de una capa de pooling se limita a combinar una serie de píxeles de entrada utilizando la función máximo, motivo por el que se denomina *max pooling*.⁶²² Por ejemplo, si queremos reducir la dimensionalidad de una imagen, podemos hacer *max pooling* 2×2 , que tesela la imagen en fragmentos de tamaño 2×2 y se queda únicamente con el máximo de cada fragmento para obtener una imagen de la mitad de ancho y de la mitad de alto, con una cuarta parte de los píxeles originales:

$$\begin{bmatrix} 4 & 14 & 15 & 1 \\ 9 & 7 & 6 & 12 \\ 5 & 11 & 10 & 8 \\ 16 & 2 & 3 & 13 \end{bmatrix} \Rightarrow \begin{bmatrix} 14 & 15 \\ 16 & 13 \end{bmatrix}$$

En una red neuronal convolutiva, las capas de pooling se suelen utilizar intercaladas entre capas convolutivas. Dado que la salida de una capa de pooling pierde gran parte de los datos de su entrada, tal vez se esté preguntando qué sentido tiene agregar las salidas de receptores de características replicados. ¿No estamos desperdiciando el esfuerzo realizado por la capa convolutiva previa a la capa de pooling? En realidad, una capa de pooling consigue dos objetivos simultáneamente, uno computacional y el otro práctico:

- La capa de pooling reduce las dimensiones de las entradas que reciben las capas posteriores de una red neuronal. Esto nos permite, con una capacidad computacional limitada, ser capaces de calcular más características distintas en paralelo, al trabajar con entradas más pequeñas.
- El uso de capas de pooling proporciona una pequeña cantidad de invarianza frente a traslaciones en la señal de entrada. El uso de pooling hace que las señales con las que trabajan las capas posteriores a una capa de pooling sean aproximadamente invariantes frente a pequeñas traslaciones en la entrada.

La invarianza (aproximada) frente a pequeñas traslaciones locales en la entrada puede resultar muy útil en la práctica. Una simple capa de max pooling 2×2 permite que, cuando la imagen de entrada aparezca desplazada un píxel, la red responda aproximadamente igual que con la

⁶²² Yi-Tong Zhou y Rama Chellappa. Computation of optical flow using a neural network. En *IEEE 1988 International Conference on Neural Networks*, volume 2, pages 71–78, 1988. DOI: 10.1109/ICNN.1988.23914

No es casualidad que se obtengan los cuatro mayores valores de la matriz 4×4 , ya que partíamos de un cuadrado mágico en el que filas, columnas y diagonales suman lo mismo (34).

imagen original sin desplazar. Esta invarianza aproximada ha resultado tener mucho éxito en múltiples aplicaciones, ya que a menudo nos interesa más ser capaces de detectar cuándo está presente alguna característica particular que determinar exactamente dónde está.

Obviamente, el uso de capas de pooling tiene un problema grave. Tras varios niveles de pooling, se pierde por completo la información acerca de la posición exacta de las cosas en la imagen de entrada. No es raro que una red diseñada para clasificar imágenes combine múltiples capas de pooling, con lo que pierde por completo la capacidad de localizar los objetos que es capaz de clasificar. Por ejemplo, si usamos 5 capas de pooling 2×2 , estamos reduciendo la resolución de las imágenes de entrada en un factor de 32 (2^5). Partiendo de imágenes de ImageNet, una base de datos de imágenes utilizada en competiciones de visión artificial, pasamos de entradas de 224×224 píxeles a arrays de características de tamaño 7×7 .

En una red convolutiva, es habitual insertar capas de pooling entre capas convolutivas consecutivas. Su función es reducir espacialmente la resolución espacial de las imágenes a la vez que se reduce el coste computacional de la red (lo que también nos permite aumentar su profundidad, el número de filtros diferentes de las capas convolutivas)

Aunque lo más habitual es utilizar *max pooling*, las unidades de una capa de *pooling* pueden aplicar otras funciones de agregación sobre sus datos de entrada, como calcular su media aritmética o su norma L2.⁶²³ Históricamente, las capas de pooling solían calcular la media [*average pooling*], aunque en la práctica se ha observado que quedarnos con el máximo suele funcionar mejor.⁶²⁴

Algunas redes convolutivas se diseñan para que utilicen pooling en unos canales pero no en otros, con el objetivo de conseguir tanto invarianza de traslación para algunas características como precisión a la hora de procesar otras características para las que no resulte válida la suposición de invarianza frente a traslaciones.⁶²⁵

El uso de *pooling* espacial facilita invarianza frente a traslaciones, pero si el pooling se realiza sobre las salidas de diferentes filtros (esto es, diferentes canales de la entrada), se puede conseguir que la red aprenda ante qué tipo de transformaciones debe ser invariante. Es la idea que subyace al uso de convoluciones 1×1 , en las que, en lugar de utilizar una función preestablecida de pooling como el máximo o la media, se aprenden los parámetros de dicha función. De ahí que una capa convolutiva con filtros 1×1 sea equivalente a una capa pooling paramétrico entre canales [*cross-channel parametric pooling*].

Se han probado múltiples estrategias de pooling en la práctica. Por ejemplo, se puede utilizar un algoritmo de clustering para seleccionar dinámicamente qué características se combinan en una capa de pooling, lo que permite definir diferentes regiones de pooling para cada imagen

⁶²³ Y-Lan Boureau, Jean Ponce, y Yann LeCun. A theoretical analysis of feature pooling in visual recognition. En *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 111–118, USA, 2010. Omnipress. ISBN 978-1-60558-907-7

⁶²⁴ Dominik Scherer, Andreas Müller, y Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. En Konstantinos Diamantaras, Wlodek Duch, y Lazaros S. Iliadis, editores, *Artificial Neural Networks – ICANN 2010*, pages 92–101, 2010. ISBN 978-3-642-15825-4. DOI: 10.1007/978-3-642-15825-4_10

⁶²⁵ Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, y Andrew Rabinovich. Going deeper with convolutions. *arXiv e-prints*, arXiv:1409.4842, 2014a. URL <http://arxiv.org/abs/1409.4842>

particular.⁶²⁶ También se puede aprender una estructura única de pooling que luego se utilice sobre todas las imágenes de la misma forma.⁶²⁷ Incluso se han propuesto estrategias de pooling más sofisticadas que pretenden ser más robustas frente a las transformaciones geométricas que se observan en imágenes reales, más allá de simples traslaciones de píxeles. Es el caso, por ejemplo de MOP [*Multi-Scale Orderless Pooling*].⁶²⁸ Este tipo de técnicas se diseñan para no tener que entrenar la red con versiones aumentadas del conjunto de entrenamiento cuando queremos que la red sea robusta frente a determinados tipos de transformaciones en su entrada.

Hiperparámetros de una capa de pooling

Las capas de pooling resumen, en su salida, la entrada que reciben. Esta agregación de entradas se traduce en que las capas de pooling contienen menos unidades que las capas convolutivas que las preceden.

En general, el pooling sólo actúa espacialmente [*spatial pooling*]. En su versión más común se limita a submuestrear la entrada espacialmente. Esto es, se utiliza el máximo como función de agregación para resumir las entradas, respetando su estructura en canales independientes. En el caso de imágenes, es común utilizar filtros de tamaño 2×2 que se aplican sobre regiones adyacentes de la entrada, sin solapamiento. Esto consigue reducir la resolución espacial de las imágenes, de $W \times H$ a $W/2 \times H/2$, con lo que se descartan directamente el 75 % de las entradas. Obviamente, aunque se descarten tres cuartas partes de las entradas, como no sabemos de antemano cuáles serán, no podemos ahorrarnos su cálculo.

El uso de pooling espacial nos obliga a especificar dos hiperparámetros de la capa de pooling al diseñar una red convolutiva:

- *Ventana [window]*

En primer lugar, hemos de especificar el tamaño de la ventana que utilizaremos para obtener cada salida de la capa de pooling; esto es, el tamaño F del filtro que aplicaremos sobre la imagen de entrada. Habitualmente, el filtro tendrá la misma extensión horizontal y verticalmente (p.ej. 2×2).

- *Paso, salto o zancada [stride]*

Igual que en las capas convolutivas, podemos especificar el tamaño S del salto que damos sobre la entrada para obtener salidas consecutivas. Lo más usual es utilizar un paso de tamaño igual que el tamaño del filtro de pooling, aunque también podemos hacer que las ventanas se solapen.

Dado que su principal misión es reducir la dimensionalidad de los datos, no es habitual utilizar *zero padding* en las capas de pooling.

⁶²⁶ Y-Lan Boureau, Nicolas Le Roux, Francis Bach, Jean Ponce, y Yann Le-Cun. Ask the locals: Multi-way local pooling for image recognition. En *2011 International Conference on Computer Vision*, pages 2651–2658, 2011. DOI: 10.1109/ICCV.2011.6126555

⁶²⁷ Yangqing Jia, Chang Huang, y Trevor Darrell. Beyond spatial pyramids: Receptive field learning for pooled image features. En *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3370–3377, 2012. DOI: 10.1109/CVPR.2012.6248076

⁶²⁸ Yunchao Gong, Liwei Wang, Ruiqi Guo, y Svetlana Lazebnik. Multi-scale orderless pooling of deep convolutional activation features. En David Fleet, Tomas Pajdla, Bernt Schiele, y Tinne Tuytelaars, editores, *Computer Vision – ECCV 2014*, pages 392–407, 2014. ISBN 978-3-319-10584-0. DOI: 10.1007/978-3-319-10584-0_26

Una capa de pooling espacial toma como entrada una imagen de tamaño $C \times W \times H$, con C canales diferentes y una resolución espacial $W \times H$. Si el tamaño del filtro de pooling es $F \times F$ y usamos S como tamaño de paso en ambas direcciones (horizontal y verticalmente), la salida de la capa convolutiva estará formada por los C canales originales, aunque su resolución espacial se reducirá a $(W - F)/S + 1$ píxeles de ancho y $(H - F)/S + 1$ píxeles de alto. En resumen, dada una entrada $C \times W \times H$, se obtiene una salida $C \times ((W - F)/S + 1) \times ((H - F)/S + 1)$.

Normalmente, sólo se utilizan dos configuraciones de capas de pooling espacial: la habitual de *max pooling* 2×2 , con $F = 2$ y $S = 2$, y una versión solapada que emplea ventanas de tamaño 3 ($F = 3$) y salto 2 ($S = 2$). En ambas, se reduce a una cuarta parte el tamaño de los datos salida con respecto a los datos de entrada. No se suelen usar ventanas de tamaño mayor porque se perderían demasiados datos: la capa de pooling resultaría demasiado destructiva.

A diferencia de otros tipos de capas que hemos visto, las capas de pooling espacial no tienen ningún parámetro que debamos ajustar durante el entrenamiento de la red neuronal, ya que se limitan a aplicar una función fija sobre sus entradas (el máximo o la media).

El pooling espacial se limita a redimensionar espacialmente los datos de entrada, preservando su profundidad (el número de canales de la señal). No obstante, también se pueden diseñar capas de pooling que reduzcan la dimensionalidad de los datos reduciendo su profundidad. Estas capas de pooling entre canales [*cross-channel pooling*] no siempre son sencillas de diseñar a priori, motivo por el que se suele recurrir al uso de capas convolutivas 1×1 capaces de ajustar automáticamente la mejor forma de combinar las señales de distintos canales.

En una red convolutiva, como las utilizadas para clasificar imágenes, es habitual que las capas convolutivas, con capas de pooling intercaladas, precedan a un conjunto de capas completamente conectadas, que esperan recibir una entrada de tamaño preestablecido. Para permitir que una red de este tipo sea capaz de trabajar con imágenes de diferentes tamaños, se pueden ajustar las regiones de pooling de forma que las capas finales de clasificación reciban siempre el mismo número de características resumidas independientemente del tamaño inicial de la imagen de entrada.

Implementación

Para implementar una capa de pooling, creamos un nuevo tipo de capa:

```
class PoolingLayer(c,w,h,k,s)
:Layer(c*w*h, m*((w-k)/s+1)*((h-k)/s+1))
```

Como en el caso de las capas convolutivas, se recibe una entrada formada por imágenes de c canales de resolución $w \times h$. La salida se generará combinando regiones de tamaño $k \times k$ usando un paso s , parámetros de los que se derivan directamente las dimensiones de la salida de la capa.

La propagación de una señal de entrada para obtener la salida de la capa es muy simple. Sólo tenemos que calcular la salida de la capa para cada región de pooling:

```
class PoolingLayer(c,w,h,k,s)
    function output = forward(input)
        for f=1:c
            for x=1:(w-k)/s+1
                for y=1:(h-k)/s+1
                    output[f][x][y] = pool(input,f,x,y);
```

La función auxiliar `pool` se encarga de realizar el resumen correspondiente a una región completa de pooling, en este caso un simple *max pooling*:

```
class PoolingLayer(c,w,h,k,s)
    function max = pool(input,x,y)
        max = -MAX_VALUE;
        for i=1:k
            for j=1:k
                pixel = input[f][s*(x-1)+i][s*(y-1)+j];
                if (pixel > max)
                    max = pixel;
```

La propagación hacia atrás del error, necesaria para poder entrenar redes multicapa, es muy sencilla en el caso de la función máximo. Sólo tenemos que enviar el gradiente del error a la entrada que generó el máximo al propagar la señal hacia adelante:

```
class PoolingLayer(c,w,h,k,s)
    function backward(error, f)
        inputDelta = 0;
        for f=1:c
            for x=1:(w-k)/s+1
                for y=1:(h-k)/s+1
                    (maxX, maxY) = max(f,x,y);
                    inputDelta[f][maxX][maxY] = error[f][x][y];
```

En este caso, la función auxiliar `max` nos devuelve los índices correspondientes a las coordenadas de donde proviene el máximo de la región de pooling (correspondientes a la entrada de la que se obtuvo la salida que dio lugar al gradiente observado del error).

Aunque podríamos implementar la función `max` de forma independiente a la función `pool` utilizada en la propagación de señales hacia adelante, es habitual aprovechar el paso hacia adelante para conservar los índices correspondientes a los máximos de cada región de pooling (a veces llamados *switches*). Sólo tenemos que modificar ligeramente nuestra implementación anterior de la función `pool`:

```
class PoolingLayer(c,w,h,k,s)
    function max = pool(input,x,y)
        max = -MAX_VALUE;
        for i=1:k
            for j=1:k
                coordX = s*(x-1)+i;
                coordY = s*(y-1)+j;
                pixel = input[f][coordX][coordY];
                if (pixel > max)
                    max = pixel;
                switchX[f][x][y] = coordX;
                switchY[f][x][y] = coordY;
```

Una vez que disponemos de los *switches*, completar la implementación de la propagación hacia atrás resulta trivial:

```
class PoolingLayer(c,w,h,k,s)
    function (maxX, maxY) = max(f,x,y)
        maxX = switchX[f][x][y];
        maxY = switchY[f][x][y];
```

De esta forma, la propagación del error hacia atrás en una capa de pooling se consigue realizar de forma más eficiente que si repitiésemos de nuevo el proceso de detectar dónde está el máximo de cada región.

Implementación para GPU

Como hemos hecho en otras ocasiones, incluimos en esta sección la versión en pseudocódigo de la implementación de una capa de pooling usando CUDA, lo que nos permite aprovechar la capacidad de cálculo paralelo de una GPU.

Todos los detalles de configuración de la capa de pooling se establecen creando un descriptor en el que se especifica el algoritmo utilizado (*max pooling*), el número de dimensiones espaciales de la entrada (2), el tamaño de las regiones de pooling (`[k k]`, correspondiente a regiones $k \times k$) y el tamaño del salto que damos al pasar de una región a la siguiente (`[s s]`, usando el mismo tamaño de salto horizontal y verticalmente). Además, si lo deseamos, podemos utilizar *zero padding*, aunque en este caso no lo hemos hecho (`[0 0]`):

```

class CUDAProxyLayer(c,w,h,k,s): PoolingLayer(c,w,h,k,s)

    function init()
        dims = 2;
        window = [k k];
        padding = [0 0];
        stride = [s s];
        cudnnCreatePoolingDescriptor(poolingDescriptor);
        cudnnSetPoolingNdDescriptor(poolingDescriptor,
            CUDNN_POOLING_MAX, CUDNN_PROPAGATE_NAN,
            dims, window, padding, stride);

    function output = forward(input)
        cudnnPoolingForward(poolingDescriptor,
            1,input,0,output);

    function backward(error)
        cudnnPoolingBackward(poolingDescriptor,
            1,output,error,0,inputDelta);

```

La implementación anterior utiliza *max pooling*, configurando el descriptor con la opción CUDNN_POOLING_MAX. La biblioteca CuDNN también dispone de una implementación de *average pooling* (usando la media en vez del máximo), para la que nos ofrece dos opciones de configuración CUDNN_POOLING_AVERAGE_COUNT_INCLUDE_PADDING y CUDNN_POOLING_AVERAGE_COUNT_EXCLUDE_PADDING, dependiendo de si queremos que los píxeles de relleno se utilicen o no a la hora de calcular la media.

En cuanto a la propagación de señales hacia adelante y del gradién-te del error hacia atrás, nos limitamos a realizar llamadas a funcio-nes especializadas de la biblioteca CuDNN que se encargan de ello: cudnnPoolingForward y cudnnPoolingBackward, respectivamente.

Críticas

El uso de pooling en una red convolutiva permite que la respuesta de la red sea aproximadamente invariante frente a pequeñas traslaciones en su entrada. Esa invarianza, aunque sea aproximada, ha resultado ser muy beneficiosa en multitud de aplicaciones prácticas de las redes convolutivas. Particularmente, en todas aquellas en las que nos interesa más detectar algo que localizarlo con precisión. De hecho, es relativamente común utilizar múltiples capas de pooling, con lo que se suele perder la posición de las cosas aun cuando la red es capaz de detectar su presencia.

Sin embargo, aunque resulte útil en la práctica para resolver problemas prácticos de ingeniería con indudable éxito, el uso de pooling no está exento de críticas. En palabras del propio Hinton, el uso de pooling “es

un gran error y el hecho de que funcionen tan bien es un desastre”.

Cuando las regiones de pooling no se solapan, el operador de pooling pierde información valiosa acerca de dónde se encuentran las cosas. Esta información resulta esencial si queremos luego ser capaces de detectar las relaciones espaciales precisas entre las distintas partes de un objeto. Para una red convolutiva convencional, una cara sigue siendo una cara aunque pongamos los ojos donde deberían estar la boca y la nariz, la boca y la nariz donde deberían estar las orejas, y las orejas donde deberían estar los ojos. Para nosotros, evidentemente, eso es cualquier cosa menos una cara.

Es cierto que si las regiones de pooling se solapan, las posiciones de las características se preservan de forma precisa. Pero, entonces, ni se gana nada usando capas de pooling desde el punto de vista computacional (no se reduce el número de operaciones necesarias), ni se está utilizando una estrategia que parezca plausible desde el punto de vista biológico.

Hay investigadores que, directamente, optan por eliminar las capas de pooling de las redes convolutivas y utilizar arquitecturas de red que consistan únicamente en capas convolutivas consecutivas.⁶²⁹ Para reducir el tamaño de las capas internas de la red, se suele recurrir a utilizar capas convolutivas en las que el tamaño del paso o salto [*stride*] sea mayor que uno.

Prescindir de las capas de pooling también ha dado buenos resultados a la hora de entrenar modelos generativos, en los que la misión de la red es sintetizar nuevos ejemplos similares a los presentes en el conjunto de entrenamiento. Es el caso de los autocodificadores variacionales [*VAEs: variational autoencoders*] o las redes con adversario generativo [*GANs: generative adversarial networks*].

El propio Hinton ha propuesto un modelo alternativo que parece algo más razonable a la hora de detectar la pose de un objeto con respecto al observador. Con el término pose se hace referencia a su posición, orientación y escala. Hinton apuesta por utilizar la misma estrategia que se utiliza en informática gráfica para sintetizar imágenes, donde una pequeña matriz, de tamaño 4×4 , se utiliza para traducir del sistema de coordenadas del objeto al sistema de coordenadas del observador. Con ella se consigue capturar fácilmente el efecto geométrico de un cambio en el punto de vista (la posición de la cámara con respecto al objeto). El problema gráfico inverso consiste en realizar la conversión inversa, del sistema de coordenadas del observador al sistema de coordenadas del objeto. La idea de Hinton, plasmada en sus redes de cápsulas [*capsule networks*] o CapsNets.⁶³⁰ Las cápsulas realizan cálculos internos sobre las entradas equivalentes a las conversiones de coordenadas comunes en informática gráfica. Esto permite que su salida sea muy informativa, dado que una cápsula local es capaz de reconocer una entidad visual desde diferentes perspectivas. No sólo eso, sino que también detecta su pose

⁶²⁹ Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, y Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv e-prints*, arXiv:1412.6806, 2014. URL <http://arxiv.org/abs/1412.6806>

⁶³⁰ Geoffrey E. Hinton, Alex Krizhevsky, y Sida D. Wang. Transforming auto-encoders. En Timo Honkela, Włodzisław Duch, Mark Girolami, y Samuel Kaski, editores, *Artificial Neural Networks and Machine Learning – ICANN 2011*, pages 44–51, 2011. ISBN 978-3-642-21735-7. doi: 10.1007/978-3-642-21735-7_6

precisa. El diseño de una red de cápsulas hace su salida equivariante, no sólo frente a traslaciones, sino también frente a rotaciones y cambios de escala. Por desgracia, su entrenamiento es mucho más costoso que el de una red convolutiva convencional.⁶³¹

El modelo de redes basadas en cápsulas, que utiliza unidades más complejas que las neuronas convencionales al incorporar una pequeña matriz de parámetros, aun siendo biológicamente poco plausible, permitiría a una red neuronal interpretar correctamente una escena aunque cambiamos de punto de vista. Simplemente, se realiza un cambio en el sistema de coordenadas utilizado y la escena se sigue interpretando exactamente de la misma forma. Es algo que nosotros realizamos con naturalidad pero que una red convolutiva convencional se muestra completamente incapaz de conseguir.

Las redes de cápsulas no son la única propuesta existente para conseguir redes con diseños alternativos. Existen algunos enfoques que pretenden ser más fieles al funcionamiento de las neuronas biológicas de nuestro córtex cerebral. Investigadores como Jeff Hawkins en Numenta (<http://numenta.com/>) y Dileep George en Vicarious (<http://www.vicarious.com/>) están explorando vías alternativas.

En el caso de Numenta, su modelo HTM [*Hierarchical Temporal Memory*] pretende modelar el funcionamiento de las columnas corticales existentes en el neocórtex. A diferencia de los modelos tradicionales de redes neuronales artificiales, que se entrena usando algoritmos iterativos basados en el gradiente descendente y *backpropagation*, las redes HTM son capaces de aprender patrones secuenciales de forma no supervisada.⁶³² Su punto fuerte es su capacidad de tratar señales con ruido y su robustez frente a variaciones en la entrada, algo que consiguen utilizando una codificación dispersa [*sparse coding*] en la que sólo el 2 % de las columnas están activas en un momento dado, una estrategia similar a la que parece utilizar nuestro cerebro para ahorrar energía. El uso comercial de las redes HTM, en cambio, es mucho más limitado que el de las redes convolutivas, pese a que, en principio, se pueden utilizar para resolver problemas de predicción, clasificación y detección de anomalías.

En definitiva, aunque su éxito actual es innegable en la resolución de problemas prácticos de ingeniería, es más que posible que las redes neuronales artificiales del futuro prescindan, de una u otra forma, de las capas de pooling.

⁶³¹ Sara Sabour, Nicholas Frosst, y Geoffrey E. Hinton. Dynamic routing between capsules. *arXiv e-prints*, arXiv:1710.09829, 2017. URL <http://arxiv.org/abs/1710.09829>

Jeff Hawkins, fundador de Palm, empresa que desarrolló asistentes digitales personales [PDAs: Personal Digital Assistants] antes de que existiesen los smartphones, creó Numenta en 2005 junto con Dileep George para hacer ingeniería inversa del neocórtex, estudiar cómo funciona el cerebro para construir máquinas que operen de acuerdo a los mismos principios que el cerebro humano.

⁶³² Jeff Hawkins. *On Intelligence*. Times Books, 2004. ISBN 0805074562

Cuestiones prácticas

Tratemos, a continuación, algunos aspectos relevantes relativos al uso en la práctica de redes convolutivas.

Arquitectura de las redes convolutivas

Una red convolutiva, CNN o ConvNet, está formada por una secuencia de capas. Todas las capas de la red transforman su entrada para generar una salida utilizando funciones diferenciables, lo que permite su entrenamiento con gradiente descendente y *backpropagation*. En general, en una red convolutiva nos encontraremos tres tipos de capas: capas convolutivas, capas de pooling y capas completamente conectadas (como las utilizadas en redes neuronales tradicionales).

La arquitectura de red convolutiva más común suele combinar capas convolutivas con capas de pooling. En ocasiones, entre la capa convolutiva y la capa de pooling se incluye una capa de unidades lineales rectificadas de tipo ReLU, que realiza una transformación no lineal de su entrada, elemento a elemento. Este patrón es tan común que, en ocasiones, las capas se contabilizan de forma diferente. Podemos contar las capas de una red convolutiva de dos formas diferentes:

- Como una red formada por capas complejas que constan de tres etapas: convolución (operación lineal), detector ReLU (función no lineal) y pooling.
- Como una red formada por capas simples consecutivas, algunas de las cuales (ReLU y pooling) ni siquiera tienen parámetros ajustables durante el entrenamiento de la red.

Esta estructura (*CONV* → *ReLU* → *POOL*) es habitual en redes convolutivas, en parte, porque nos permite establecer una analogía entre la topología de la red neuronal artificial y la estructura del córtex visual primario V1. En el córtex visual, primero se crean mapas espaciales (los mapas de características de las capas convolutivas), a continuación se genera cierta actividad neuronal en células “simples” (los detectores ReLU) y, por último, la actividad de células “complejas” proporciona cierta invarianza con respecto a pequeños cambios de posición (pooling espacial) y de iluminación (pooling entre canales [*cross-channel pooling*]).

A menudo, se prescinde por completo de los rectificadores y las redes convolutivas se construyen intercalando capas convolutivas con capas de pooling, utilizando un patrón (*CONV* → *POOL*). En este caso, dado que las capas convolutivas son capas lineales, la no linealidad de la red la aportan las capas de pooling. También es habitual encontrarse redes convolutivas en las que aparecen varias capas convolutivas consecutivas antes de cada capa de pooling,^{633,634} con una estructura de tipo (*CONV* → *CONV* → *POOL*).

Cuando queremos resolver problemas de clasificación, por ejemplo, se apilan múltiples capas para formar la arquitectura de la red convolutiva completa. En particular, puede que nos encontremos con dos topologías alternativas:

Esta arquitectura del córtex visual también sirve de inspiración para otros modelos de *pooling* como el realizado por unidades de tipo *maxout*, que realizan *max pooling* entre canales diferentes.

Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, y Yoshua Bengio. Maxout networks. En *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1319–1327, 2013b. URL <http://jmlr.org/proceedings/papers/v28/goodfellow13.html>

⁶³³ Alex Krizhevsky, Ilya Sutskever, y Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. En *NIPS’2012 Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL <https://goo.gl/Ddt8Jb>

⁶³⁴ Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, y Li Fei-Fei. Large-scale video classification with convolutional neural networks. En *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014. DOI: 10.1109/CVPR.2014.223

- Redes convolutivas para clasificación en las que, durante sus primeras etapas, se utilizan múltiples capas convolutivas usando un patrón ($CONV \rightarrow POOL$) o ($CONV \rightarrow CONV \rightarrow POOL$), tras las cuales se recurre a capas convencionales completamente conectadas [FC , *fully-connected*], que suelen terminar en una capa de tipo *softmax*. La arquitectura resultante suele ser, en forma de expresión regular, $(CONV^+ \rightarrow POOL)^+ \rightarrow FC^* \rightarrow SOFTMAX$.
- Redes convolutivas para clasificación por regiones en las que las capas completamente conectadas finales se sustituyen por una capa convolutiva con un mapa de características por clase y una capa de pooling promedio [*average pooling*] que proporciona un único valor por clase que se puede utilizar directamente como entrada de un clasificador softmax. La idea es que la capa convolutiva con un mapa por clase sea capaz de aprender cómo de probable es que una clase esté presente en cada región espacial de la entrada de la red. La arquitectura resultante será de la forma $(CONV^+ \rightarrow POOL)^+ \rightarrow CONV(\text{por clase}) \rightarrow AVGPOOL \rightarrow SOFTMAX$.

Ambas alternativas nos permiten convertir una señal estructurada de entrada (p.ej. en forma de imágenes multicanal) en una distribución de clases que estima la probabilidad de que la entrada corresponda a cada clase concreta de nuestro problema de clasificación.

Al entrenar la red, ajustaremos los parámetros de las capas *CONV* y *FC*, ya que las capas de pooling *POOL* (y de rectificación *ReLU*, en caso de que las empleemos) implementan funciones preestablecidas sin parámetros.

Al diseñar la red, tendremos que fijar valores concretos para los hiperparámetros de las diferentes capas. Además de fijar las dimensiones de las capas completamente conectadas (*FC* y *SOFTMAX* a la salida de la red), tendremos que definir los distintos hiperparámetros de las capas convolutivas (*CONV*) y de pooling (*POOL*).

Una arquitectura de red convolutiva típica es de la forma $((CONV^+ \rightarrow ReLU?)^N \rightarrow POOL?)^M \rightarrow FC^K \rightarrow SOFTMAX$, donde ? indica los componentes opcionales. Usualmente, $0 < N \leq 3$, $M > 0$ y $0 < K \leq 3$. Las etapas iniciales, convolutivas y de pooling, van reduciendo la dimensión espacial de la imagen hasta que ésta tiene un tamaño pequeño. En ese momento, se pasa a una red multicapa convencional, que se encarga de realizar la clasificación de la imagen de entrada.

En redes más profundas, es habitual concatenar dos capas convolutivas antes de cada capa de pooling, con el objetivo de que sean capaces de extraer características más complejas de la entrada antes de realizar la operación destructiva de pooling. Aunque varias capas convolutivas consecutivas siempre se podrían reemplazar por una única capa, se prefiere apilar capas convolutivas con filtros pequeños antes que usar

una única capa cuyos filtros sean de mayor tamaño. Si apilamos dos capas convolutivas 3×3 , la primera tiene un campo receptivo local de tamaño 3×3 , mientras que la segunda cubre una extensión 5×5 con respecto a la entrada de la primera capa. Si apilásemos una tercera capa, cada una de sus neuronas tendría un campo receptivo de tamaño 7×7 . En la práctica, se suele preferir entrenar 3 capas con filtros de tamaño 3×3 en lugar de una única capa de tamaño 7×7 . En el primer caso, para una señal de C canales, tendríamos $3 \times C(3 \times 3 \times C) = 27C^2$ parámetros, mientras que necesitaríamos $C(7 \times 7 \times C) = 49C^2$ para una capa convolutiva 7×7 . Además, si en las capas introducimos no linealidades de tipo ReLU, podemos extraer características más complejas de la entrada que las que una capa lineal sería capaz de identificar. Estas ventajas no se obtienen gratis, ya que puede que necesitemos más espacio en memoria para mantener los resultados intermedios de las distintas capas convolutivas cuando entrenemos la red con gradiente descendente y *backpropagation*.

Aunque la descrita hasta el momento sea la arquitectura más común de las redes convolutivas, competiciones como ImageNet han dado como resultado topologías de red alternativas como Inception,⁶³⁵ de Google, o las redes de residuos ResNets,⁶³⁶ de Microsoft Research Asia.

En cuanto a los hiperparámetros concretos con los que configurar las capas individuales de una red convolutiva, se pueden utilizar algunos criterios de tipo heurístico:

- El tamaño de la capa de entrada, normalmente, se escoge para que sea divisible por dos múltiples veces (con el objetivo de incluir múltiples capas de pooling 2×2). Algunos valores habituales son 28 (MNIST), 32 (CIFAR-10), 64, 96 (STL-10), 224 (ImageNet), 384 o 512, donde entre paréntesis aparecen conjuntos de datos estándar de esas dimensiones.
- Las capas convolutivas suelen utilizar filtros pequeños de tamaño 3×3 ($F = 3$) o, como mucho, 5×5 ($F = 5$). Si se emplean filtros más grandes, suele ser sólo en la primera capa de la red, aunque normalmente se prefiere aplicar capas convolutivas con filtros más pequeños. En las capas convolutivas casi siempre se emplea un paso o *stride* $S = 1$ y se suele llenar la entrada con ceros para que no se modifiquen las dimensiones de la entrada ($Z = (F - 1)/2$).
- Las capas de pooling que reducen espacialmente las dimensiones de los datos suelen submuestrear la entrada utilizando *max pooling* 2×2 ($F = 2$) sin solapamiento entre las regiones de pooling ($S = 2$). Cuando se trabaja con imágenes, cada capa de pooling 2×2 reduce a un cuarto el número de píxeles que las etapas siguientes han de procesar (se descarta el 75 % de los valores de entrada).

Los criterios anteriores simplifican la selección de hiperparámetros

⁶³⁵ Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, y Andrew Rabinovich. Going deeper with convolutions. En *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9, 2015a. ISBN 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298594

⁶³⁶ Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. Deep residual learning for image recognition. En *CVPR'2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. DOI: 10.1109/CVPR.2016.90

de una red convolutiva. El uso de *padding* en las capas convolutivas nos permite eliminar problemas de dimensionamiento (se preservan las dimensiones espaciales de las entradas) y deja a las capas de pooling la misión de reducir espacialmente las dimensiones de las señales con las que trabaja la red.

Un efecto similar se podría conseguir si utilizásemos un tamaño de salto mayor que 1 en las capas convolutivas, aunque suele funcionar mejor la combinación de convolución seguida de pooling espacial. De esta forma, además, asignamos responsabilidades a las diferentes capas con mayor claridad: las capas de pooling son las encargadas de modular las dimensiones espaciales de la entrada, mientras que las capas convolutivas son las responsables de manipular su profundidad (el número de filtros que contienen).

Preservar las dimensiones de la entrada usando *zero padding*, además de facilitarnos la conexión de unas capas con otras, nos permite aprovechar al máximo la información que puede contener los bordes de la entrada. Sin ese relleno, las dimensiones espaciales de la entrada se irían reduciendo paulatinamente tras cada convolución, los bordes se irían perdiendo y tendríamos que ser extremadamente cuidadosos a la hora de dimensionar las diferentes capas.

En ocasiones, las decisiones concretas las tomaremos en función de los recursos computacionales de los que dispongamos, ya que la cantidad de memoria necesaria se puede disparar con facilidad. Por este motivo es habitual que la primera capa convolutiva de la red utilice filtros más grandes, ya que esto nos permitirá reducir el tamaño de los arrays internos y el consumo de memoria de la red. Por ejemplo, para la primera capa, podríamos utilizar filtros 7×7 y *stride* 2 (como en ZFNet) o filtros 11×11 y *stride* 4 (como en AlexNet). Ambas redes están diseñadas para trabajar con ImageNet, base de datos de imágenes $224 \times 224 \times 3$. Si, sobre esas imágenes, usásemos una capa convolutiva con 64 filtros 3×3 y *padding* 1 para preservar sus dimensiones, necesitaríamos $64 \times 224 \times 224 \times 3$ neuronas en la primera capa, casi 10 millones. Como al entrenar la red necesitamos tanto activaciones como gradientes (8 bytes por neurona), necesitaremos 72MB de memoria sólo para la primera capa. En cambio, la primera capa de ZFNet reduce las dimensiones de la entrada a imágenes de 113×113 y la primera capa de AlexNet las deja en 57×57 . De esta forma, una capa de 64 filtros necesitaría menos de dos millones y medio de neuronas siguiendo la estrategia de ZFNet y apenas seiscientos mil en el caso de AlexNet (un ahorro considerable de memoria, casi 4x en ZFNet y 16x en AlexNet).

El uso de memoria suele ser el cuello de botella que determina la arquitectura concreta de una red convolutiva. Dado que las GPU actuales suelen disponer de algunos gigabytes de memoria, hay que planificar detalladamente el consumo de memoria de nuestra red:

- Necesitamos reservar espacio para mantener los niveles de activación de todas las neuronas de la red (así como sus gradientes cuando la estemos entrenando, lo que duplica el espacio necesario). Las primeras capas, de mayor resolución, serán normalmente a las que dediquemos mayores recursos. Cuando sólo estemos usando la red, no entrenándola, en principio podríamos reducir el espacio de memoria necesario almacenando únicamente los valores correspondientes a la capa actual (sólo necesitaremos memoria para almacenar dos arrays, las entrada y salidas de una capa particular).
- Necesitamos almacenar los parámetros de la red. En este caso, las capas convolutivas suelen tener un número reducido de parámetros y la parte del león corresponde a las capas finales de la red, completamente conectadas.
- Por último, no debemos olvidar que, cuando estemos entrenando la red, también necesitaremos memoria para almacenar los gradientes de los parámetros de la red y toda la información adicional que sea necesaria para la implementación del algoritmo de optimización que estemos utilizando (momentos y caché para las medias móviles usadas en algoritmos como AdaDelta o RMSProp).

Obviamente, si queremos aprovechar el paralelismo de la GPU para entrenar la red usando minilotes, tendremos que ajustar el número de muestras de cada minilote en función de la memoria que tengamos disponible. Podemos asumir que cada valor almacenado en memoria se representa en coma flotante usando 4 bytes, de donde podemos obtener los requisitos en memoria de nuestra red convolutiva completa, expresados en gigabytes. Si hemos sido demasiado ambiciosos sobre el papel, no queda otra que, en el mejor caso, reducir el tamaño del minilote o, en el peor caso, las dimensiones de nuestra red convolutiva hasta que pueda entrar en la memoria de nuestra GPU.

Para finalizar esta sección sobre el diseño arquitectónico de las capas convolutivas, un par de comentarios sobre el uso de sesgos en las capas convolutivas. Típicamente, tal como hicimos en nuestra implementación, se suele incluir un sesgo por canal de salida, sesgo que se comparte entre todas las neuronas de un mapa de características. En las capas localmente conectadas, sin embargo, cada neurona individual suele tener su propio sesgo. En el caso intermedio, el de la convolución en mosaico [*tiled convolution*], lo normal es que se comparta el sesgo entre las neuronas que utilizan el mismo kernel de convolución (una forma de garantizar que el número de parámetros depende del número de filtros, no de las dimensiones de la entrada de la red).

Una red convolutiva bien diseñada puede aceptar entradas de diferente tamaño si ajustamos dinámicamente los tamaños de sus regiones de

pooling para mantener constante el tamaño de su salida. En algunas ocasiones, no obstante, puede que nos interese que el tamaño de la salida también sea variable. Una característica clave de las capas convolutivas (y también de las capas de pooling) es que, una vez entrenadas para detectar patrones, se pueden utilizar directamente sobre entradas de diferentes tamaños. Es una consecuencia inmediata del hecho de que los parámetros de las capas convolutivas no están vinculados a entradas concretas, como en las redes neuronales convencionales, sino a las máscaras de convolución. Esto nos facilita aplicar directamente una red convolutiva (o, al menos, su parte realmente convolutiva) sobre entradas de tamaño arbitrario. Si variamos las dimensiones espaciales de la entrada, sólo cambiará el número de convoluciones realizadas, que depende de las dimensiones de la entrada. En este caso, el tamaño de salida de la red convolutiva variará de forma dinámica conforme modifiquemos las dimensiones de su entrada, lo que puede ser útil en aplicaciones en las que queramos etiquetar los píxeles de la entrada o procesar una imagen (p.ej. eliminar ruido o transferir el estilo de una imagen a otra).

Entrenamiento de redes convolutivas

En una red convolutiva, el uso de conexiones sinápticas locales que definen campos receptivos locales y de pesos compartidos que permiten detectar las mismas características en distintas posiciones de la entrada dotan a las capas convolutivas de la red de equivarianza (las salidas varían en la misma dirección que las entradas). Además, las redes convolutivas incorporan capas de pooling diseñadas de tal forma que versiones ligeramente modificadas de la entrada producen salidas similares, al menos cuando la modificación consiste en un pequeño desplazamiento. Esta “invarianza por estructura”, derivada de la propia topología de la red, dota a las redes convolutivas de cierta robustez en la práctica, aunque no siempre resulta suficiente. Por no mencionar que el propio diseño de la red introduce un sesgo [*bias*], en el sentido genérico del término en aprendizaje automático (el correspondiente a la descomposición del error en sesgo y varianza). Es decir, mediante decisiones de diseño arquitectónico de la red estamos incorporando en nuestro modelo prejuicios acerca de la forma particular de resolver un problema, prejuicios que puede que no siempre sean acertados.

Durante el entrenamiento de una red neuronal, también es posible dotar a la red de un grado mayor de invarianza frente a las transformaciones de los datos de entrada que nos esperamos encontrar en la práctica mediante la ampliación del conjunto de entrenamiento. En el conjunto de entrenamiento ampliado se incluirán versiones transformadas de los ejemplos existentes en el conjunto de entrenamiento original. Podemos introducir ejemplos con ruido (si queremos que nuestra red sea robusta

a señales con ruido), ejemplos rotados (si no siempre nos llegarán las entradas perfectamente alineadas) o ejemplos deformados para reproducir cambios en el punto de vista (por ejemplo, si estamos entrenando un robot móvil para reconocer objetos desde diferentes posiciones). El uso de conjuntos de entrenamiento ampliados supone, como es lógico, que el entrenamiento de la red puede resultar mucho más costoso computacionalmente. A cambio, nos permite conseguir redes más robustas en la práctica y el proceso de optimización utilizado en su entrenamiento puede llegar a descubrir formas novedosas de utilizar los parámetros de las redes neuronales multicapa. Formas que, por el lado positivo, a nosotros nunca se nos habrían llegado a ocurrir y, por el lado negativo, nunca sepamos realmente cómo interpretar.

El entrenamiento en sí de una red convolutiva es completamente análogo al de cualquier red neuronal multicapa. Sólo tenemos que calcular el gradiente del error con respecto a los kernels de convolución dado el gradiente del error con respecto a las salidas de la capa convolutiva.⁶³⁷ Las capas de pooling, como carecen de parámetros, no tienen mucho que entrenar. Sólo tienen que ser capaces de propagar el error hacia atrás.

En el entrenamiento de redes convolutivas, el uso de técnicas heurísticas como la normalización por lotes puede tener un impacto notable sobre el rendimiento de la red una vez entrenada. En el caso de la normalización por lotes, es importante que apliquemos la misma normalización en las distintas localizaciones espaciales de un mapa de características, para que los estadísticos de cada mapa de características sean los mismos independientemente de la posición concreta.

La normalización por lotes no es la única estrategia de normalización que se ha evaluado para redes convolutivas. Algunas se han propuesto con la intención de implementar los mecanismos de inhibición lateral que se observan en los cerebros biológicos y tienen nombres tan peculiares como capas de normalización de respuesta local [*LRN, Local Response Normalization*] presentes en AlexNet. En el caso de una capa LRN, cada entrada se divide por $(K + (\alpha/n) \sum x^2)^\beta$, donde n es el tamaño de la región local, K es típicamente 2, la sumatoria se realiza sobre la región centrada en esa entrada (con *zero padding* donde resulte necesario) y su funcionamiento se parametriza mediante los hiperparámetros α y β . Aunque nos podamos encontrar capas similares a esta en algunas herramientas software de *deep learning*, su uso no es muy común ya que su impacto en la práctica es, por ser benevolentes, no demasiado significativo.

Antes de que se diseñasen estrategias que facilitasen el entrenamiento de redes profundas con muchas capas ocultas (como la normalización por lotes), era habitual entrenar las redes capa a capa, utilizando un algoritmo *greedy*. El entrenamiento por capas [*layer-wise pretraining*] no requiere realizar una propagación completa de la señal a través de toda la red en

⁶³⁷ Ian J. Goodfellow. Technical report: Multidimensional, Downsampled Convolution for Autoencoders. Technical report, Université de Montréal, 2010

cada iteración del algoritmo de aprendizaje (ni de la señal de entrada hacia adelante, ni del gradiente del error hacia atrás). El entrenamiento por capas de una red convolutiva⁶³⁸ consiste en entrenar la primera capa de forma aislada, lo que nos permite extraer un primer conjunto de características de la señal de entrada. Una vez que ya tenemos la primera capa, utilizamos esas características como entrada para entrenar una segunda capa con respecto a esas características. Y así sucesivamente hasta alcanzar el número de capas deseado.

La misma estrategia se puede utilizar para pre-entrenar una red convolutiva profunda entrenando primero una red convolutiva. Las capas de esta red convolutiva, una vez entrenada, se pueden utilizar para inicializar las primeras y últimas capas de una red más profunda. Por ejemplo, podríamos comenzar entrenando una red con 11 capas para después utilizar sus cuatro primeras capas y sus tres últimas para inicializar redes más profundas, de hasta 19 capas, tal como se hizo para crear la red VGG19.⁶³⁹

Incluso es posible pre-entrenar redes convolutivas sin utilizar aprendizaje supervisado. La estrategia más rudimentaria consiste, simplemente, en diseñar a mano las características que deseamos que la red convolutiva sea capaz de extraer, igual que en el neocognitrón de Fukushima. Si sabemos que determinadas características pueden ser relevantes para nuestro problema, podemos incluir detectores de esas características en las capas convolutivas (p.ej. aristas con diferentes orientaciones) para después entrenar una red multicapa convencional. Incluso podemos probar a utilizar distintos conjuntos de filtros aleatorios y quedarnos con el que mejor funcione.⁶⁴⁰ Es una estrategia económica a la hora de elegir la arquitectura de una red convolutiva: primero evaluamos varias redes convolutivas entrenando únicamente su última capa y, luego, nos quedamos con la mejor de ellas para utilizar ya un proceso más costoso de entrenamiento que nos permita ajustar sus parámetros. No es la única estrategia que podemos utilizar, ya que también podríamos optar por emplear técnicas de aprendizaje no supervisado para descubrir características potencialmente útiles, como el algoritmo de las k medias.⁶⁴¹ Llevado este enfoque al extremo, podemos llegar a entrenar una red convolutiva sin llegar a realizar una sola convolución durante el proceso de entrenamiento completo de la red.^{642,643} De esta forma, podemos entrenar redes muy grandes sin incurrir en el coste computacional que supone realizar convoluciones (coste que sólo asumiremos a la hora de realizar inferencias, al usar la red ya entrenada).

El pre-entrenamiento no supervisado de redes convolutivas era popular hasta 2013, época en la que los conjuntos de datos etiquetados eran relativamente pequeños y la potencia computacional de los ordenadores era más limitada que ahora. Actualmente, lo habitual es entrenar las redes convolutivas utilizando el gradiente descendente y *backpropagation*,

⁶³⁸ Honglak Lee, Roger Grosse, Rajesh Ranganath, y Andrew Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. En *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 609–616, 2009. ISBN 978-1-60558-516-1. DOI: 10.1145/1553374.1553453

⁶³⁹ Karen Simonyan y Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR'2015*, arXiv:1409.1556, 2014. URL <http://arxiv.org/abs/1409.1556>

⁶⁴⁰ Andrew M. Saxe, Pang Wei Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, y Andrew Y. Ng. On random weights and unsupervised feature learning. En *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 1089–1096. Omnipress, 2011. ISBN 978-1-4503-0619-5

⁶⁴¹ Adam Coates, Andrew Y. Ng, y Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. En *AISTATS'2011 Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 215–223, 2011. URL <https://goo.gl/Uyp1yZ>

⁶⁴² Marc'Aurelio Ranzato, Fu Jie Huang, Y-Lan Boureau, y Yann LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. En *CVPR'2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007. DOI: 10.1109/CVPR.2007.383157

⁶⁴³ Koray Kavukcuoglu, Pierre Sermanet, Y-Lan Boureau, Karol Gregor, Michael Mathieu, y Yann L. Cun. Learning convolutional feature hierarchies for visual recognition. En *NIPS'2010 Advances in Neural Information Processing Systems 23*, pages 1090–1098, 2010. URL <https://goo.gl/1ZjGrU>

como cualquier otra red neuronal multicapa. Eso, cuando las entrenamos nosotros, que muchas veces ni hace falta....

Aprendizaje por transferencia

En Inteligencia Artificial, el aprendizaje por transferencia consiste en aplicar el conocimiento que hayamos obtenido en un dominio particular para resolver problemas en otro dominio, generalmente relacionado con el primero. En el caso de las redes convolutivas, podemos recurrir a una red que haya sido previamente entrenada para resolver problemas genéricos (como clasificar imágenes de la base de datos ImageNet) para después emplearla en la resolución de un problema más especializado (detectar objetos particulares en imágenes). Esto puede resultar especialmente útil si no disponemos de demasiados datos etiquetados para nuestro problema particular. Aun cuando el conjunto de datos del que disponemos no sea suficiente para entrenar adecuadamente una red completa, podemos inicializar nuestro modelo con los pesos de una red previamente entrenada y ajustarlos en un proceso de afinado [*fine-tuning*] para mejorar su funcionamiento en nuestro problema particular. De forma alternativa, podemos utilizar, tal cual, los pesos de una red ya entrenada para extraer características que luego utilizaremos para entrenar únicamente las últimas capas de nuestra red especializada.

Es fácil conseguir las configuraciones de redes ya entrenadas con millones de imágenes, como las que participan en competiciones como ImageNet Large Scale Visual Recognition Challenge (ILSVRC), <http://www.image-net.org/>. Los autores de muchas de esas redes las suelen poner a disposición del público con licencias de uso flexibles. Eso nos permite usar, como punto de partida para la resolución de nuestro problema particular, las redes que representan el estado del arte en visión artificial, como AlexNet (2012), ZFNet (2013), Inception (2014), VGG (2014), ResNet (2015) o Xception (2016). Incluso hay versiones optimizadas, que requieren menos memoria, para su uso en dispositivos móviles, como es el caso de SqueezeNet (2016) o MobileNet (2017). Diferentes versiones de estas redes, ya entrenadas, suelen estar disponibles en las herramientas de *deep learning*, como en el Caffe Model Zoo o las “aplicaciones” de Keras (<https://keras.io/applications>).

A partir de una red genérica de clasificación de imágenes, podemos utilizar aprendizaje por transferencia para resolver problemas relacionados como detectar objetos en vehículos autónomos, transferir estilos de una imagen a otra o generar descripciones textuales de imágenes y videos, entre otras muchas aplicaciones en las que las redes convolutivas se han utilizado con éxito. En la mayor parte de los problemas en los que las redes convolutivas se pueden utilizar, podemos partir de una red que funcione bien en ImageNet, descargar de Internet el modelo

Andrej Karpathy nos recomienda que no intentemos ser héroes. Rara vez resulta necesario diseñar y entrenar una red convolutiva desde cero.

Andrej Karpathy. CS231n: Convolutional Neural Networks for Visual Recognition. Stanford, 2017. URL <http://cs231n.github.io/>

pre-entrenado y adaptarlo a nuestras necesidades específicas afinándolo con los datos de los que dispongamos.

A todos aquéllos que padecan el síndrome NIH [*not invented here*], sólo hay que recordarles que normalmente no dispondrán de un conjunto de datos lo suficientemente grande como para entrenar una red convolutiva completa desde cero, partiendo de pesos inicializados aleatoriamente. Una red pre-entrenada con ImageNet, que contiene 1.2 millones de imágenes de miles de clases diferentes, puede utilizarse de diferentes formas:

- *Como extractor de características*

Se coge la red entrenada con ImageNet, se elimina(n) la(s) última(s) capa(s) completamente conectada(s) y se utiliza el resto como mecanismo de extracción de características para nuestro conjunto de datos. Dada una imagen, usamos las salidas de las capas que tomamos prestadas de la red pre-entrenada, a veces denominadas códigos CNN, las empleamos como entradas para entrenar una red multicapa convencional para resolver nuestro problema particular. También podemos usarlas para entrenar cualquier otro modelo de aprendizaje automático (como una SVM o un clasificador softmax). Sólo es importante recordar que, si los que diseñaron la red utilizaron esas salidas rectificadas (con unidades ReLU), como suele ser habitual, también las rectifiquemos nosotros para que el rendimiento de nuestro sistema sea óptimo.

- *Para afinar el modelo pre-entrenado*

Entrenar una red convolutiva moderna sobre ImageNet puede necesitar varias semanas utilizando múltiples GPU. En vez de comenzar desde cero nuestra red, podemos usar los pesos de la red previamente entrenada para inicializar los pesos de nuestra red, que luego ajustaremos de la forma tradicional usando el gradiente descendente y *backpropagation*. Podemos afinar todas las capas de la red o, lo más común, mantener fijas las iniciales (para evitar problemas de sobreaprendizaje) y ajustar sólo las capas posteriores de la red. La idea tras esta estrategia consiste en aprovechar los detectores de características genéricos de las capas iniciales, entrenados sobre millones de imágenes, y afinar únicamente los detectores más específicos de las capas finales, aquellos que contienen detalles propios de la estructura del problema y que, tal vez, no se ajusten demasiado bien a nuestras necesidades particulares.

Andrej Karpathy recomienda decidir nuestra estrategia de aprendizaje por transferencia en función del tamaño de nuestro conjunto de datos y de su similitud con el conjunto de datos con el que se entrenó la red original. Por ejemplo, ImageNet contiene imágenes naturales de miles de objetos, tal como nos los podríamos encontrar por la calle. Sin embargo,

ImageNet contiene clases diferentes para múltiples razas de perro, algo que puede no resultar demasiado relevante para el problema particular que nos interese a nosotros.

si nuestra aplicación trabajará con imágenes de tipo médico (radiografías, imágenes microscópicas...) puede no tener demasiado sentido que usemos aprendizaje por transferencia y, en una situación así, tal vez sí tengamos que entrenar una red convolutiva desde cero. Resumiendo las recomendaciones de Karpathy:

- Conjunto de datos pequeño y similar al original: Tal vez no sea una buena idea intentar ajustar los parámetros del modelo pre-entrenado. Dado que esperamos que las características identificadas en la red original sean relevantes en nuestro problema, mejor entrenar un clasificador a partir de los códigos CNN de la red original.
- Conjunto de datos grande y similar al original: Al disponer de más datos, podemos ajustar los parámetros de la red original sin temer demasiado que el resultado final sufra de los problemas derivados del sobreaprendizaje (más probable cuanto menor sea nuestro conjunto de datos).
- Conjunto de datos pequeño y muy diferente al original: No disponemos de datos suficientes para poder entrenar una red convolutiva compleja, por lo que tendremos que conformarnos con entrenar un clasificador sencillo. Puede que incluso debamos ignorar el modelo pre-entrenado por completo, ya que probablemente no nos aporte demasiado en este caso, al extraer características que tienen poco que ver con nuestro problema. Como mucho, sólo podremos aprovechar las primeras capas del modelo pre-entrenado.
- Conjunto de datos grande y muy diferente al original: Es el único caso en el que podemos/debemos permitirnos entrenar una red convolutiva desde cero. En la práctica, no obstante, puede que inicializar los pesos de nuestra red con los de la red original, aunque tenga poco que ver, facilite el entrenamiento de nuestro modelo (mejor eso que inicializar los pesos aleatoriamente).

El inconveniente de utilizar redes pre-entrenadas es que su diseño arquitectónico restringe nuestras opciones de diseño. Aunque se pueden reutilizar capas convolutivas para trabajar con imágenes de diferente resolución, no podemos extraer arbitrariamente capas de una red pre-entrenada.

A la hora de ajustar los parámetros de una red previamente entrenada, suelen emplearse tasas de aprendizaje más pequeñas que las que utilizaríamos para ajustar un modelo partiendo de pesos inicializados aleatoriamente. Es algo lógico si partimos de la suposición de que los parámetros de la red pre-entrenada ya son relativamente buenos: no queremos distorsionarlos demasiado, sólo adaptarlos gentilmente a nuestro problema particular.

Se han realizado diversos estudios empíricos para evaluar hasta qué punto funciona bien el aprendizaje por transferencia en redes convolutivas. Los resultados son, por lo general, bastante positivos.^{644,645,646}

Visualización de las redes convolutivas

En la práctica, puede resultar útil visualizar gráficamente el funcionamiento de una red convolutiva. La visualización puede centrarse en diferentes aspectos de la red, en función de qué es lo que pretendamos resaltar:

- *Visualización de los niveles de activación de las neuronas*

La forma más sencilla de visualizar el funcionamiento de una red convolutiva es representar gráficamente los niveles de actividad de los distintos mapas de características de una red convolutiva como respuesta a una entrada dada. Cuando la red se entrena correctamente, cada mapa tiende a especializarse en identificar características concretas de la imagen de entrada. En las primeras capas de la red, las neuronas responderán a características locales de la imagen, como aristas/fronteras entre regiones o zonas de un color determinado. Capas posteriores tal vez lleguen a distinguir características más elaboradas de forma selectiva, como la presencia de una cara o de algún objeto particular.

Este tipo de visualización puede servir para identificar mapas de características que no muestran actividad alguna para muchas entradas diferentes, lo que puede indicar que sus filtros no se han entrenando correctamente, tal vez por haber utilizado tasas de aprendizaje demasiado elevadas.

- *Visualización de los filtros de convolución*

Otra forma de visualizar una red convolutiva ya entrenada es mostrar los pesos de sus máscaras de convolución. En la primera capa convolutiva de la red, que trabaja directamente con los píxeles de la imagen, esta visualización puede permitirnos identificar qué característica concreta de las imágenes es capaz de detectar.

Una red bien entrenada mostrará filtros bien definidos, sin demasiado ruido. La presencia de ruido en los kernels de convolución puede ser consecuencia de un entrenamiento insuficiente o de no haber regularizado bastante la red, lo que conduce a situaciones de sobreaprendizaje en las que la red se adapta en exceso a las peculiaridades del conjunto de entrenamiento.

En ocasiones, la visualización de las características detectadas por una red neuronal tal vez no resulte demasiado informativa, pero puede llegar a ser realmente llamativa visualmente, mostrándonos cómo

⁶⁴⁴ Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, y Trevor Darrell. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. *arXiv e-prints*, arXiv:1310.1531, 2013. URL <http://arxiv.org/abs/1310.1531>

⁶⁴⁵ Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, y Stefan Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. *arXiv e-prints*, arXiv:1403.6382, 2014. URL <http://arxiv.org/abs/1403.6382>

⁶⁴⁶ Jason Yosinski, Jeff Clune, Yoshua Bengio, y Hod Lipson. How transferable are features in deep neural networks? En *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3320–3328. MIT Press, 2014. URL <https://goo.gl/AZacCB>

consigue una red convolutiva identificar aristas, texturas, patrones, partes y objetos. Eso sí, la visualización resultante puede ser un tanto psicodélica.⁶⁴⁷

- *Visualización de las entradas de la red*

Una visualización más interesante para intentar interpretar qué causa determinadas respuestas de la red consiste en utilizar la red sobre un conjunto grande de imágenes (p.ej. todo el conjunto de entrenamiento) para localizar ante qué imagen una respuesta particular es máxima.⁶⁴⁸ También existe la posibilidad de reconstruir las entradas que causan una activación determinada en los niveles internos de la red, sintetizando muestras no existentes en el conjunto de entrenamiento mediante un proceso de deconvolución.⁶⁴⁹

Al ver qué características tienen las imágenes que corresponden a diferentes activaciones, podemos llegar a enterarnos qué es lo que detecta realmente la red. Eso no quiere decir que la red sea capaz de identificar categorías semánticas, como podríamos pensar ingenuamente, sino únicamente que la red tiende a responder de cierta manera ante la presencia de ciertos patrones en su señal de entrada. Esto puede parecer poco intuitivo para los legos en la materia, pero resulta esencial comprender que, aunque la respuesta de la red a una entrada sea la “correcta”, su respuesta a una entrada ligeramente diferente puede ser completamente distinta.⁶⁵⁰

Una forma sencilla de detectar si la red está respondiendo realmente a características relevantes de su entrada consiste en comparar su salida con la imagen completa y con la misma imagen en la que ocluimos, parcial o totalmente, los objetos de nuestro interés. Si la red sigue dando la misma respuesta es que no está respondiendo a lo que supuestamente deseamos identificar.

Si vamos variando la posición de la oclusión, podemos construir un mapa de calor en el que se muestre la probabilidad de que se identifique correctamente el objeto en función de la posición de la oclusión. Esto es, vamos borrando diferentes regiones de la imagen y observamos la salida de la red para, en función de si cambia o no, generar un mapa de calor. Ese mapa de calor nos indica, en cierto modo, qué regiones de la imagen de entrada son las que más contribuyen a la salida de la red. Visualizaciones de este tipo⁶⁵¹ pueden ayudarnos a determinar si la red está identificando la imagen correctamente al fijarse en las características que realmente son relevantes, o su proceso de aprendizaje le ha llevado a descubrir correlaciones en el conjunto de entrenamiento que difícilmente le permitirán generalizar correctamente. Como en el clásico ejemplo de los tanques, en el que la red aprendía a identificar la presencia de tanques camuflados en una imagen, pero no porque apareciese un tanque, sino porque todas las imágenes con tanques se habían tomado en días nublados. Si en

⁶⁴⁷ Chris Olah, Alexander Mordvintsev, y Ludwig Schubert. Feature visualization. *Distill*, 2017. DOI: 10.23915/distill.00007. URL <https://distill.pub/2017/feature-visualization>

⁶⁴⁸ Ross B. Girshick, Jeff Donahue, Trevor Darrell, y Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CVPR'2014*, arXiv:1311.2524v5, 2014a. URL <http://arxiv.org/abs/1311.2524>

⁶⁴⁹ Matthew D. Zeiler y Rob Fergus. Visualizing and understanding convolutional networks. *arXiv e-prints*, arXiv:1311.2901, 2013. URL <http://arxiv.org/abs/1311.2901>

⁶⁵⁰ Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, y Rob Fergus. Intriguing properties of neural networks. *ICLR'2014*, arXiv:1312.6199, 2014b. URL <http://arxiv.org/abs/1312.6199>

⁶⁵¹ Konda Reddy Mopuri, Utsav Garg, y R. Venkatesh Babu. CNN fixations: An unraveling approach to visualize the discriminative image regions. *arXiv e-prints*, arXiv:1708.06670, 2017. URL <http://arxiv.org/abs/1708.06670>

nuestro conjunto de datos de entrenamiento aparecen sesgos de este tipo, puede que terminemos con un modelo que puede que sea perfecto para discriminar el fondo de la imagen, pero resultará completamente inútil para su supuesto propósito.

Obviamente, como en cualquier otro problema de minería de datos, también podemos intentar recurrir a técnicas de reducción de la dimensionalidad. Un algoritmo habitual de este tipo es t-SNE (*t-distributed stochastic neighbor embedding*),⁶⁵² una variante del método SNE propuesto por Hinton unos años antes.⁶⁵³ Una red convolutiva puede interpretarse como una transformación que, paulatinamente, va generando una representación en la que las diferentes clases sean linealmente separables. Para generar una representación bidimensional de una red convolutiva de clasificación, podemos extraer los códigos CNN (las salidas de las capas anteriores a la capa final que clasifica las imágenes) y pasárselos como entrada a un método de reducción de la dimensionalidad como t-SNE para obtener vectores bidimensionales para cada imagen. Entonces, podemos combinar las imágenes en una especie de mosaico en el que las distancias de una imagen a otra en dos dimensiones corresponden a su distancia en la representación multidimensional de los códigos CNN. El resultado, aunque no resulte especialmente útil para interpretar el funcionamiento de la red convolutiva, sí que puede ser visualmente atractivo.

Seguridad de las redes convolutivas

El comportamiento de una red neuronal multicapa puede ser extremadamente no lineal. Esto hace que la red proporcione salidas completamente diferentes ante entradas ligeramente distintas. Un actor malintencionado podría, entonces, manipular la entrada de la red de forma prácticamente imperceptible para conseguir que se equivocase drásticamente. No sólo es que una red neuronal pueda proporcionar una salida correcta por las razones equivocadas, es que podemos conseguir que una salida aparentemente correcta dé resultados completamente erróneos.

En ocasiones, las modificaciones en la señal de entrada serán consecuencia de ruido y generarán errores aparentemente aleatorios en el funcionamiento de la red neuronal. Alguien podría poner una pegatina, aparentemente inofensiva, sobre una señal de tráfico y conseguir despistar por completo a un vehículo autónomo a la hora de interpretar correctamente la señal, por más que la red neuronal encargada de clasificar imágenes de señales de tráfico la hayamos entrenado con múltiples condiciones de visibilidad, de iluminación o desde muy diferentes perspectivas.

En situaciones más delicadas, sin embargo, la manipulación de la entrada puede ser intencionada y tener consecuencias desastrosas. Imagine qué puede suceder si alguien manipula una señal de tráfico impercepti-

⁶⁵² Laurens van der Maaten y Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. URL <http://www.jmlr.org/papers/v9/vandermaaten08a.html>

⁶⁵³ Geoffrey E. Hinton y Sam T. Roweis. Stochastic neighbor embedding. En *NIPS'2002 Advances in Neural Information Processing Systems 15*, pages 857–864. MIT Press, 2002. URL <https://goo.gl/hkqjR3>

blemente para el ojo humano. Para la red neuronal, la imagen de entrada podría interpretarse de forma radicalmente opuesta, lo que podría poner en serios apuros a un vehículo autónomo, además de poner en riesgo la vida de sus ocupantes. Una forma más de “crimen perfecto” que podría resultar casi indetectable.

En sistemas de detección basados en señales biométricas, como detectores faciales o escáneres de iris, el uso de redes neuronales para tomar una decisión de autentificación del usuario también está sujeto a los mismos problemas de seguridad, mostrando una vez más que la seguridad siempre es relativa (la seguridad completa nunca existe).

Todos los sistemas basados en el uso de técnicas de *deep learning* son, en principio, vulnerables ante ejemplos diseñados por un adversario.⁶⁵⁴ El diseño de esos ejemplos es casi trivial si conocemos los detalles internos de la red o del conjunto de entrenamiento.⁶⁵⁵ Pero también se puede lograr sin tener conocimiento alguno de la red.⁶⁵⁶ Basta con poder tantejar el funcionamiento de la red remota, como si de una caja negra se tratase, y entrenar un modelo localmente usando ejemplos diseñados por el adversario y etiquetados por la red remota. Con esta estrategia se puede conseguir que cualquier red se equivoque en más del 90 % de los ejemplos diseñados maliciosamente. En el caso extremo, podríamos lograr que una red se equivoque con sólo manipular un píxel de su entrada.⁶⁵⁷

En sistemas físicos del mundo real, tal vez no tengamos acceso directo a la red ni a sus predicciones. Sólo podemos analizar el comportamiento externo de un vehículo autónomo, pero no acceder a sus módulos internos. Aun así, se puede conseguir que un sistema se equivoque en el sentido que a nosotros nos interese.⁶⁵⁸

A la hora de diseñar sistemas que utilicen internamente redes neuronales, es importante que el diseñador sea consciente de sus limitaciones. De esta forma podrá tomar contramedidas que mitiguen en parte los riesgos de seguridad a los que está expuesto cualquier sistema basado en el uso de técnicas de aprendizaje automático. Por ejemplo, puede entrenar sus modelos utilizando conjuntos de entrenamiento ampliados o mejorar la robustez del sistema ante ciertos tipos de ataques utilizando técnicas de entrenamiento basadas en el uso de modelos con adversario. O, simplemente, introducir ruido en la entrada del sistema cuando se está usando para contrarrestar, sobre la marcha, el impacto de manipulaciones cuidadosamente diseñadas por un adversario.⁶⁵⁹ Una combinación adecuada de contramedidas puede conseguir que los ejemplos diseñados por un adversario acaben siendo una curiosidad académica más que una amenaza seria a la seguridad del sistema.

Aplicaciones

La neurociencia computacional,⁶⁶⁰ también conocida como neurocién-

⁶⁵⁴ Patrick McDaniel, Nicolas Papernot, y Z. Berkay Celik. Machine learning in adversarial settings. *IEEE Security & Privacy*, 14(3):68–72, 2016. ISSN 1540-7993. DOI: 10.1109/MSP.2016.51

⁶⁵⁵ Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, y Ananthram Swami. The limitations of deep learning in adversarial settings. En *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 372–387, 2016. ISBN 978-1-5090-1751-5. DOI: 10.1109/EuroSP.2016.36

⁶⁵⁶ Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, y Ananthram Swami. Practical black-box attacks against machine learning. En *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS ’17*, pages 506–519, 2017. ISBN 978-1-4503-4944-4. DOI: 10.1145/3052973.3053009

⁶⁵⁷ Jiawei Su, Danilo Vasconcellos Vargas, y Kouichi Sakurai. One pixel attack for fooling deep neural networks. *arXiv e-prints*, arXiv:1710.08864, 2017. URL <http://arxiv.org/abs/1710.08864>

⁶⁵⁸ Alexey Kurakin, Ian J. Goodfellow, y Samy Bengio. Adversarial examples in the physical world. *arXiv e-prints*, arXiv:1607.02533, 2016. URL <http://arxiv.org/abs/1607.02533>

⁶⁵⁹ Abigail Graese, Andras Rozsa, y Terrance E. Boult. Assessing threat of adversarial examples on deep neural networks. En *15th IEEE International Conference on Machine Learning and Applications, ICMLA 2016, Anaheim, CA, USA, December 18-20, 2016*, pages 69–74, 2016. ISBN 978-1-5090-6167-9. DOI: 10.1109/ICMLA.2016.0020

⁶⁶⁰ Michael A. Arbib y James J. Bonaiuto, editores. *From Neuron to Cognition via Computational Neuroscience*. MIT Press, 2016. ISBN 0262034964

cia teórica o matemática, es la rama de la neurociencia que pretende describir el funcionamiento del cerebro desde un punto de vista computacional. Desde los primeros intentos de modelar la visión,⁶⁶¹ se ha recurrido a modelos conexionistas en los que se compartían parámetros para diferentes localizaciones espaciales. Esos modelos, como el neocognitrón de Fukushima,⁶⁶² intentaban ser fieles a lo que se conocía sobre el funcionamiento del cerebro, si bien carecían de algoritmos de entrenamiento con los que pudiesen ser utilizados en la resolución de problemas reales mediante la utilización de técnicas de aprendizaje automático.

Los modelos diseñados por neurocientíficos y, en particular, los trabajos del británico David Marr en el MIT,⁶⁶³ tuvieron un impacto notable en el desarrollo de las técnicas de visión por computador. La visión por computador, o visión artificial, es una especialidad dentro de Inteligencia Artificial que intenta automatizar las tareas realizadas por nuestro sistema visual, de forma que un ordenador sea capaz de llegar a comprender el contenido de imágenes y vídeos. A diferencia de la informática gráfica, especializada en la síntesis de imágenes fotorrealistas, la visión artificial se encarga de analizar imágenes.

Según Marr, que falleció prematuramente antes de poder ver publicado su libro, la visión es una tarea computacional que consigue generar una representación interna del estímulo visual recibido a través de la retina. La visión estereoscópica nos permite construir una representación que, aun no llegando a ser tridimensional, sí incorpora información de profundidad en lo que Marr denominaba *sketch 2½D*, de dos dimensiones y media. Esta observación permitió a Marr descomponer, desde el punto de vista algorítmico, la visión en tres niveles, que correspondían a la construcción de un *sketch* primario (2D), el *sketch 2½D* y, finalmente, el *sketch 3D* que contiene la representación tridimensional del estímulo visual. Para Marr, esta descomposición proporcionaba un sistema formal completo para describir la visión.

Determinar exactamente cómo, a partir de la imagen captada por la retina, se detectan, procesan y representan las características de una escena desde el punto de vista algorítmico sigue siendo un problema no resuelto al que dedican sus esfuerzos multitud de neurocientíficos y especialistas en Inteligencia Artificial. Desde un punto de vista algorítmico, el principal obstáculo es el problema inverso: la imposibilidad de conocer la realidad del mundo físico únicamente a partir de las imágenes captadas por la retina (natural o artificial). Ese problema inverso, que la evolución biológica ha resuelto en la práctica con notable éxito, está aún lejos del alcance de las técnicas actuales de visión artificial.

Tradicionalmente, las técnicas de visión artificial se han basado en diseñar múltiples métodos de ingeniería de características. Estos métodos ayudan a extraer características útiles de las imágenes de entrada, como

⁶⁶¹ David Marr y Tomaso Poggio. Cooperative computation of stereo disparity. *Science*, 194(4262):283–287, 1976. ISSN 0036-8075. DOI: 10.1126/science.968482

⁶⁶² Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980. ISSN 1432-0770. DOI: 10.1007/BF00344251

⁶⁶³ David Marr. *Vision*. W.H. Freeman, 1983. ISBN 0716715678

Mientras que una sola cámara es capaz de captar una imagen 2D, en un sistema de visión estereoscópica, dos cámaras perciben imágenes ligeramente distintas de la escena, lo que les permite, mediante un proceso de emparejamiento, determinar la profundidad de los objetos en la escena. Observe que la percepción estereoscópica no proporciona directamente un modelo tridimensional de la escena (como los modelos con los que se trabaja en informática gráfica).

- SIFT [*scale-invariant feature transform*],⁶⁶⁴
- HoG [*histogram of oriented gradients*],⁶⁶⁵
- GLOH [*Gradient Location and Orientation Histogram*],⁶⁶⁶
- textones [*textons*] para la percepción de texturas,⁶⁶⁷ o
- imágenes de espines [*spin images*] para reconocer superficies y objetos en escenas 3D,⁶⁶⁸

así como innumerables variantes y generalizaciones de las anteriores.

Pese a décadas de esfuerzo intentando diseñar características a medida, la victoria de una red convolutiva diseñada por un estudiante de doctorado en una competición de visión artificial, puso patas arriba el campo de la visión artificial en 2012. El uso de redes convolutivas, capaces de aprender a diseñar sus propios filtros de convolución, resulta mucho menos tedioso que el diseño manual de mecanismos de extracción de características. Además, pese a sus limitaciones formales (muy criticadas hasta entonces), mejora los resultados obtenidos con otras técnicas de visión artificial.

Veamos algunas de sus aplicaciones más notables, algunas de ellas anteriores a la revolución del *deep learning*...⁶⁶⁹

Clasificación de imágenes

Un problema típico de visión artificial es la clasificación de imágenes, identificar imágenes en las que aparece un objeto de interés de otras imágenes en las que no aparece. Para construir un clasificador basado en redes neuronales o cualquier otra técnica de aprendizaje automático, se necesita un conjunto de datos de entrenamiento adecuado.

Gran parte del éxito de las técnicas de *deep learning* proviene de la disponibilidad de grandes conjuntos de datos. Estos conjuntos de datos han permitido que las redes neuronales actuales sean capaces de alcanzar al ser humano a la hora de resolver determinados problemas que, hasta hace no demasiado tiempo, sólo eran capaces de resolver para casos de juguete.

Una red neuronal puede conseguir una precisión aceptable al clasificar imágenes si se dispone de miles de imágenes de cada categoría en la que queremos clasificar las imágenes. Cuando se dispone de millones de imágenes, su precisión iguala e incluso supera a la del ser humano en esta tarea.

El problema de la clasificación de imágenes es tan común, que existen conjuntos de datos estándar sobre los que se suele evaluar el rendimiento de cada nueva técnicas que se propone. Algunos de los conjuntos de datos más conocidos que se utilizan, a modo de *benchmark*, son los siguientes:

- *MNIST [Modified NIST]*⁶⁷⁰
- <http://yann.lecun.com/exdb/mnist/>

⁶⁶⁴ David G. Lowe. Object recognition from local scale-invariant features. En *Proceedings of the 7th IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157, 1999. DOI: 10.1109/ICCV.1999.790410

⁶⁶⁵ Navneet Dalal y Bill Triggs. Histograms of oriented gradients for human detection. En *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893, 2005. DOI: 10.1109/CVPR.2005.177

⁶⁶⁶ Krystian Mikolajczyk y Cordelia Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(10):1615–1630, 2005. ISSN 0162-8828. DOI: 10.1109/TPAMI.2005.188

⁶⁶⁷ Bela Julesz. Textons, the elements of texture perception, and their interactions. *Nature*, 290(5802):91–97, March 1981. ISSN 0028-0836. DOI: 10.1038/290091a0

⁶⁶⁸ Andrew Johnson. *Spin-Images: A Representation for 3-D Surface Matching*. PhD thesis, Carnegie Mellon University, August 1997

⁶⁶⁹ Yann LeCun, Koray Kavukcuoglu, y Clément Farabet. Convolutional networks and applications in vision. En *ISCAS'2010 - Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 253–256, 2010. DOI: 10.1109/ISCAS.2010.5537907

⁶⁷⁰ Yann LeCun, Leon Bottou, Yoshua Bengio, y Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998a. ISSN 0018-9219. DOI: 10.1109/5.726791

Una versión preprocesada del conjunto de datos original, recopilado por una agencia estadounidense, el NIST [*National Institute of Standards and Technology*]. Esta base de datos consiste en decenas de miles de imágenes escaneadas de dígitos manuscritos. Es tan común su uso que Geoffrey Hinton la describe como “la *drosophila* del aprendizaje automático”, la versión informática de la mosca de la fruta que utilizan los biólogos para realizar sus experimentos en un entorno controlado de laboratorio. Cada imagen, en escala de grises, contiene 28×28 píxeles: 784 píxeles que se han estudiado minuciosamente desde finales de los años 80, cuando Yann LeCun comenzó el desarrollo de las redes convolutivas. El conjunto de entrenamiento contiene 60000 imágenes de dígitos escritos por 250 personas diferentes (la mitad empleados del censo estadounidense, la mitad estudiantes de secundaria). El conjunto de test contiene 10000 imágenes creadas por otras 250 personas, diferentes a las del conjunto de entrenamiento (aunque de los mismos grupos demográficos).

Si se cansa de utilizar este conjunto de datos, tal vez le interese echarle un vistazo a otro conjunto de datos con el mismo formato y tamaño, aunque esta vez para clasificar prendas de ropa en lugar de dígitos del 0 al 9: *Fashion MNIST*, creado por Zalando (<https://github.com/zalandoresearch/fashion-mnist>).

- *SVHN [Street View House Numbers]*⁶⁷¹
<http://ufldl.stanford.edu/housenumbers/>
- *CIFAR [Canadian Institute for Advanced Research]*⁶⁷²
<https://www.cs.toronto.edu/~kriz/cifar.html>

Los conjuntos de datos CIFAR-10 y CIFAR-100 también contienen imágenes en color de 32×32 píxeles. En este caso, CIFAR es un organismo canadiense que mantuvo la financiación de la investigación en redes neuronales artificiales cuando éstas se encontraban en su peor momento. Alex Krizhevsky, un doctorando de Hinton, etiquetó estos dos conjuntos a partir de una base de datos de 80 millones de pequeñas imágenes no etiquetadas, provenientes de la web. El primero, CIFAR-10, contiene 10 clases diferentes de vehículos y animales, mientras que el segundo, CIFAR-100, contiene imágenes de 100 clases diferentes agrupadas en 20 superclases (desde mamíferos acuáticos y alimentos hasta muebles y obras de ingeniería).

- *STL*⁶⁷³
<https://cs.stanford.edu/~acoates/stl10/>

⁶⁷¹ Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, y Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. En *NIPS'2011 Workshop on Deep Learning and Unsupervised Feature Learning*, 2011. URL http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf

⁶⁷² Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>

⁶⁷³ Adam Coates, Andrew Y. Ng, y Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. En *AISTATS'2011 Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 215–223, 2011. URL <https://goo.gl/Uyp1yZ>

El conjunto de datos STL-10 incluye un número menor de imágenes etiquetadas por clase que CIFAR-10. Como aquél, contiene 10 clases de animales y vehículos (cambiando las ranas de CIFAR-10 por monos), si bien las imágenes son de mayor resolución, 96×96 píxeles. Los 9216 píxeles de cada imagen, frente a los 1024 de CIFAR, sirven para poner a prueba la escalabilidad de los algoritmos de aprendizaje. Además, STL incluye 100000 imágenes no etiquetadas para evaluar técnicas de aprendizaje no supervisado.

- *NORB [NYU Object Recognition Benchmark]*⁶⁷⁴
<http://www.cs.nyu.edu/~ylclab/data/norb-v1.0/>

No todos los conjuntos de datos de imágenes utilizan los tres canales típicos de las imágenes en color. El conjunto de datos NORB contiene 97200 pares de imágenes binoculares de 50 figuritas de juguete agrupadas en 5 categorías (animales, figuras humanas, aviones, camiones y coches). Cada par contiene dos imágenes en escala de grises de 96×96 píxeles que corresponden a una imagen estereoscópica de figuritas vistas desde distintos ángulos y bajo diferentes condiciones de iluminación.

- *GTSRB [German Traffic Sign Recognition Benchmark]*⁶⁷⁵
<http://benchmark.ini.rub.de/?section=gtsrb>

La base de datos GTSRB está diseñada para evaluar algoritmos de visión artificial capaces de diferenciar entre diferentes señales de tráfico. En lugar de presentar las imágenes en un formato normalizado, ya preprocessado para facilitar su uso, las imágenes varían en resolución, de 15×15 a 250×250 , tal como las percibe un coche autónomo cuando circula por la carretera. Aun así, una red convolutiva profunda es capaz de reconocer correctamente las señales un 99.46% de las veces, mejor que un ser humano.⁶⁷⁶

- *LFW [Labeled Faces in the Wild]*⁶⁷⁷
<http://vis-www.cs.umass.edu/lfw/>

Los sistemas de reconocimiento facial presentan algunas peculiaridades que los hacen diferentes de los demás problemas de clasificación que hemos mencionado. En concreto, un sistema de este tipo ha de ser capaz de reconocer correctamente una clase sin disponer de miles de ejemplos de la clase. De hecho, habitualmente, sólo disponen de un único ejemplo por clase, la imagen de la persona a la que han de identificar correctamente.

LFW en un pequeño conjunto de datos en comparación con otros, con poco más de trece mil imágenes. Eso sí, las imágenes son de 5749 personas diferentes (de las cuales sólo 1680 aparecen en más de una fotografía).

- *ImageNet*⁶⁷⁸
<http://www.image-net.org/>

⁶⁷⁴ Yann LeCun, Fu Jie Huang, y Leon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. En *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 2, pages II–97–104, 2004. DOI: 10.1109/CVPR.2004.1315150

⁶⁷⁵ Johannes Stallkamp, Marc Schlipsing, Jan Salmen, y Christian Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. En *IEEE International Joint Conference on Neural Networks*, pages 1453–1460, 2011. DOI: 10.1109/IJCNN.2011.6033395

⁶⁷⁶ Dan C. Ciresan, Ueli Meier, Jonathan Masci, y Jürgen Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012a. ISSN 0893-6080. DOI: 10.1016/j.neunet.2012.02.023. Selected Papers from IJCNN 2011

⁶⁷⁷ Gary B. Huang, Manu Ramesh, Tamara Berg, y Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007. URL <http://vis-www.cs.umass.edu/lfw/lfw.pdf>

⁶⁷⁸ Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, y Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. En *CVPR'2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. DOI: 10.1109/CVPR.2009.5206848. URL http://www.image-net.org/papers/imagenet_cvpr09.pdf

Conjunto de datos	Categorías	Imágenes	Resolución
MNIST	10	70000	$28 \times 28 \times 1$
SVHN	10	630420	$32 \times 32 \times 3$
CIFAR-10	10	60000	$32 \times 32 \times 3$
CIFAR-100	100	60000	$32 \times 32 \times 3$
STL-10	10	13000	$96 \times 96 \times 3$
NORB	5	97200	$96 \times 96 \times 2$
GTRSB	>50	>50000	$\leq 250 \times 250 \times 3$
LFW	5749	13233	$250 \times 250 \times 3$
ImageNet	21841	14197122	$\geq 224 \times 224 \times 3$
Places	476	7076580	$\geq 200 \times 200 \times 3$
Places2	434	10624928	$\geq 200 \times 200 \times 3$
COCO	91	328000	$\geq 256 \times 256 \times 3$

Con permiso de MNIST, la más conocida de las bases de datos de imágenes, por su utilización en la competición internacional ILSVRC [*ImageNet Large Scale Visual Recognition Competition*]. Esta base de datos contiene millones de imágenes de más de veinte mil clases diferentes, que corresponden a los nombres recogidos por la ontología WordNet (más concretamente, a conjuntos de sinónimos o *synsets*).

- *Places*^{679,680}
<http://places.csail.mit.edu/>, <http://places2.csail.mit.edu/>

Una base de datos menos conocida que ImageNet, con 7 millones de imágenes etiquetadas que corresponden a distintos tipos de escenas (lugares en los que se tomó cada fotografía). A diferencia de ImageNet, que está orientada hacia la detección de objetos en imágenes, Places está ideada para reconocer el contexto de una imagen. Su segunda versión incluye más de 10 millones de imágenes de más de 400 categorías diferentes. Las categorías corresponden, no a objetos presentes en la imagen, sino a categorías semánticas que definen su función (p.ej. se distingue dormitorio de habitación de hotel).

- *COCO [Common Objects in Context]*⁶⁸¹
<http://cocodataset.org/>

Otro conjunto de datos utilizado en competiciones de visión artificial en el que se incluyen imágenes de escenas complejas de la vida diaria en las que aparecen objetos comunes en su contexto natural. Las imágenes corresponden a 91 tipos de objetos que un niño de 4 años podría reconocer con facilidad. Se utiliza, por ejemplo, para ver hasta qué punto un ordenador es capaz de describir textualmente el contenido de una imagen, algo que, por ahora, los niños hacen mejor sin demasiada dificultad.

Los resultados que se han conseguido con los conjuntos de datos

Tabla 7: Algunos conjuntos de datos estándar para evaluar técnicas de clasificación de imágenes. El número de imágenes indica el número total de imágenes etiquetadas del conjunto de datos (hoy en día es sencillo conseguir millones de imágenes no etiquetadas en Internet).

⁶⁷⁹ Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, y Aude Oliva. Learning deep features for scene recognition using places database. En *NIPS'2014 Advances in Neural Information Processing Systems 27*, pages 487–495, 2014. URL <https://goo.gl/wkSGRu>

⁶⁸⁰ Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, y Antonio Torralba. Places: A 10 million Image Database for Scene Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP(99):1–1, 2017. ISSN 0162-8828. DOI: 10.1109/TPAMI.2017.2723009

⁶⁸¹ Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, y C. Lawrence Zitnick. Microsoft COCO: common objects in context. *arXiv*, arXiv:1405.0312, 2014b. URL <http://arxiv.org/abs/1405.0312>

anteriores dan una muestra de la evolución de las técnicas de clasificación de imágenes basadas en redes neuronales y de las heurísticas que han permitido mejorar su rendimiento hasta llegar a ser, en muchos casos, más precisas y fiables que los seres humanos realizando las mismas tareas.

MNIST

Yan LeCunn trabajó durante años en el problema de reconocimiento óptico de caracteres manuscritos. Su red neuronal convolutiva, LeNet-5,⁶⁸² es capaz de clasificar los dígitos de MNIST con una tasa de error de sólo el 0.82 %. Con una red multicapa convencional, podríamos bajar del 2 % de error, pero no mucho más allá sin utilizar otro tipo de estrategias. Para conseguir este resultado, la red de LeCunn utiliza dos capas convolutivas seguidas de dos capas de submuestreo (*max pooling* espacial), tras las que se añaden tres capas convencionales completamente conectadas. De la entrada, formada por dígitos de $32 \times 32 \times 1$ píxeles, se extraen seis mapas de características $28 \times 28 \times 6$, que se reducen mediante pooling $14 \times 14 \times 6$. La siguiente capa convolutiva genera dieciséis mapas de características $10 \times 10 \times 16$ que, nuevamente, se reducen mediante pooling a mapas $5 \times 5 \times 16$. La salida de esta última capa se emplea como entrada de una red multicapa convencional. Aprovechando su conocimiento sobre el problema, LeCunn diseñó una red capaz de procesar millones de cheques diarios en Estados Unidos. Su red es muy robusta ante la presencia de ruido y diferentes deformaciones en las imágenes de entrada, gracias a lo que podríamos llamar “invarianza por estructura”.

El éxito de LeCunn en los años 90 hizo que las redes convolutivas se utilizasen con éxito en la resolución de problemas de OCR [*Optical Character Recognition*], demostrando ser especialmente versátiles en el reconocimiento de textos manuscritos, un problema mucho más difícil que el OCR sobre texto impreso o mecanografiado. El secreto de su rendimiento consistía, básicamente, en disponer de conjuntos de datos lo más amplios posibles.⁶⁸³ Para ello no hacía falta disponer de cantidades enormes de datos etiquetados, sino que bastaba con, dado un conjunto de entrenamiento, generar de forma sintética otros ejemplos de entrenamiento deformando las imágenes originales de la misma forma en que podrían aparecer deformadas bajo diferentes condiciones reales. Es la estrategia utilizada por Dan Ciresan y sus colaboradores de IDSIA en Suiza para reducir la tasa de error en MNIST al 0.35 %. No hay que recurrir a técnicas sofisticadas de entrenamiento ni arquitecturas novedosas de red. Basta con ampliar el conjunto de entrenamiento y emplear una red convolutiva convencional. De esta forma se consigue lo que podríamos denominar “invarianza por entrenamiento”.

Jugando un poco con el diseño arquitectónico de la red convolutiva también se pueden conseguir algunas mejoras. Es el caso de las redes de

⁶⁸² Yann LeCun, Leon Bottou, Yoshua Bengio, y Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998a. ISSN 0018-9219. doi: 10.1109/5.726791

⁶⁸³ Patrice Y. Simard, Dave Steinkraus, y John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. En *ICDAR'2003 Proceedings of the 7th International Conference on Document Analysis and Recognition*, pages 958–963, Aug 2003. doi: 10.1109/ICDAR.2003.1227801

cápsulas de Hinton,⁶⁸⁴ con las que la tasa de error en MNIST se reduce hasta el 0.25 %, u otras redes diseñadas por Ciresan,⁶⁸⁵ con las que el error se queda sólo en el 0.23 %. Para reducir aún más la tasa de error, los investigadores suelen recurrir al uso de ensambles.

Números en Google Street View...

Una aplicación a gran escala de las redes convolutivas es la interpretación de los números que aparecen en los portales de las casas en Google Street View, lo que permite a Google afinar su servicio de mapas corrigiendo errores que pueden tener otras fuentes de datos a partir de las cuales se obtiene la numeración de las calles. Los investigadores de Google se propusieron lograr una precisión equivalente a la de un ser humano etiquetando imágenes, que ellos estimaron en un 98 % y conseguir una cobertura del 95 % (ser capaces de etiquetar automáticamente el 95 % de las imágenes). Utilizaron la cobertura como métrica principal de rendimiento del sistema, manteniendo fijo su nivel de precisión requerido.

Usando una red convolutiva convencional se encontraron que la cobertura del sistema no llegaba al 90 %, muy lejos de su objetivo final. Sin embargo, observaron que la tasa de error del sistema era la misma tanto en el conjunto de entrenamiento como en el conjunto de prueba. Es decir, el sistema aprendía a generalizar de forma correcta pero era su entrenamiento el que fallaba. Observando las imágenes que se le suministraban a la red, se dieron cuenta de que muchas de ellas aparecían recortadas de más, con fragmentos de dígitos eliminados por una etapa de preprocessamiento anterior. Consiguieron una mejora del 10 % en la cobertura del sistema, simplemente, con aumentar la región recortada de las imágenes originales que se utilizaba para entrenar la red convolutiva. Las últimas décimas de rendimiento del sistema las consiguieron afinando los hiperparámetros de su red neuronal.⁶⁸⁶

... y gatos en YouTube

Aunque no tenga el impacto comercial de Google Street View, el primer proyecto realizado en Google Brain consistió en suministrarle a una red profunda un montón de vídeos de YouTube.⁶⁸⁷ A partir de 10 millones de imágenes de 200×200 píxeles, se entrenó una red neuronal de 9 capas con mil millones de conexiones. Para ello hizo falta una implementación asíncrona del gradiente descendente estocástico que se ejecutó, durante 3 días, en un cluster de mil ordenadores con 16000 núcleos. Este proyecto sirvió de demostración de que una red puede aprender a extraer características de imágenes aunque las imágenes no hayan sido previamente etiquetadas (un costoso proceso manual). Esas características permitían identificar gatos en vídeo de YouTube y, posteriormente, se utilizaron para entrenar una red que conseguía

⁶⁸⁴ Sara Sabour, Nicholas Frosst, y Geoffrey E. Hinton. Dynamic routing between capsules. *arXiv e-prints*, arXiv:1710.09829, 2017. URL <http://arxiv.org/abs/1710.09829>

⁶⁸⁵ Dan C. Ciresan, Ueli Meier, y Jürgen Schmidhuber. Multi-column deep neural networks for image classification. En *CVPR'2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, 2012b. DOI: 10.1109/CVPR.2012.6248110

⁶⁸⁶ Ian Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, y Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. En *ICLR'2014 International Conference on Learning Representations*, 2014a. URL <https://arxiv.org/abs/1312.6082>

⁶⁸⁷ Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S. Corrado, Jeff Dean, y Andrew Y. Ng. Building high-level features using large scale unsupervised learning. En *Proceedings of the 29th International Conference on International Conference on Machine Learning*, ICML'12, pages 507–514, 2012. ISBN 978-1-4503-1285-1. URL <https://icml.cc/2012/papers/73.pdf>

una precisión del 15.8 % a la hora de reconocer categorías de objetos en ImageNet. En esta base de datos, con 20000 categorías diferentes, ¡la precisión esperada eligiendo al azar sería sólo del 0.005 %! En ese momento, el mejor resultado publicado era un humilde 9.5 %, aunque la situación estaba a punto de cambiar...

ImageNet

ImageNet es una base de datos de millones de imágenes clasificadas de acuerdo a la taxonomía de conceptos de WordNet. Se usa, anualmente, como banco de pruebas de una competición internacional llamada ILSVRC [*ImageNet Large Scale Visual Recognition Challenge*].⁶⁸⁸ Hasta 2011, la competición venía siendo dominada por investigadores en visión artificial que hacían un uso intensivo de técnicas de extracción de características diseñadas a mano (ingeniería de características). Sin embargo, en 2012 pasó algo que cambió por completo el panorama. Desde entonces, la competición la dominan las redes neuronales artificiales y su éxito en ella es, en gran medida, responsable del auge del *deep learning*.

- *AlexNet*⁶⁸⁹

En 2012, Alex Krizhevsky, un estudiante de doctorado bajo la supervisión de Geofrey Hinton en la Universidad de Toronto, entrenó una red neuronal sobre las 1.3 millones de imágenes de tamaño $224 \times 224 \times 3$ del conjunto de entrenamiento de ImageNet LSVRC-2010, que entonces contenía 1000 clases diferentes. Sobre el conjunto de prueba consiguió tasas de error del 39.7 % (top-1) y del 18.9 % (top-5). En competiciones de este tipo, en las que hay miles de clases diferentes, se considera aceptable un clasificador que incluya la clase correcta entre sus 5 primeras respuestas.

Los resultados de Krizhevsky eran significativamente mejores que los obtenidos por otros investigadores hasta el momento. En la competición de ese año, que ganó claramente, consiguió una tasa de error del 15.3 % frente al 26.1 % de su más inmediato perseguidor (nada menos que un 40 % de reducción en la tasa de error con respecto al segundo clasificado). Su red neuronal, bautizada como AlexNet, incluía 60 millones de parámetros y medio millón de neuronas, con un total de 7 capas ocultas. AlexNet contiene cinco capas convolutivas, algunas de las cuales iban seguidas de capas de *max-pooling*, y dos capas completamente conectadas antes de su capa softmax final. Su arquitectura, por tanto, era muy similar a LeNet, aunque más profunda, más grande y con varias capas convolutivas consecutivas (anteriormente, era habitual intercalar siempre una capa de pooling entre las capas convolutivas).

Como preprocessamiento de los datos, Krizhevsky se limitó a restarle, a cada píxel, la media de los valores de los píxeles en las imágenes del

⁶⁸⁸ Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, y Fei-Fei Li. ImageNet Large Scale Visual Recognition Challenge. *arXiv e-prints*, arXiv:1409.0575, 2014. URL <http://arxiv.org/abs/1409.0575>

⁶⁸⁹ Alex Krizhevsky, Ilya Sutskever, y Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. En *NIPS'2012 Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL <https://goo.gl/Ddt8Jb>

conjunto de entrenamiento. A la hora de entrenar la red, eso sí, empleó múltiples trucos para mejorar su capacidad de generalización, entre los que se incluye la ampliación del conjunto de entrenamiento y *dropout* como técnica de regularización. Como algoritmo de optimización, utilizó el gradiente descendente estocástico con momentos y *weight decay*. El proceso le llevó menos de una semana utilizando dos GPU en paralelo.

- **ZFNet**⁶⁹⁰

Al año siguiente, la red convolutiva creada por Matthew Zeiler y Rob Fergus, después apodada ZFNet [Zeiler & Fergus Net], consiguió reducir la tasa de error del 15.3 % del año anterior al 12.5 %, si bien es cierto que la competición ILSVRC'2013 la ganaron ensambles de modelos con los que el error bajó hasta el 11.2 % (también del equipo Clarifai de Zeiler).

La arquitectura de ZFNet era similar a AlexNet, ajustando algunos de sus hiperparámetros. En particular, aumentando el tamaño de sus capas convolutivas intermedias y reduciendo el tamaño de los filtros de la primera capa (11×11 con *stride* 4 en AlexNet, 7×7 con *stride* 2 en ZFNet), lo que les permite retener más información de los píxeles originales de las imágenes.

Zeiler y Fergus también investigaron cómo visualizar el funcionamiento de su red mediante “deconvoluciones”, con el objetivo de que “el desarrollo de nuevos modelos no se reduzca a un proceso de prueba y error”. Las deconvoluciones permiten pasar de características a píxeles en imágenes, lo contrario que las convoluciones, lo que permite describir (y, tal vez, explicar) el funcionamiento interno de las redes convolutivas mediante la visualización de qué tipos de entradas activan un mapa de características determinado.

- **GoogLeNet (Inception)**⁶⁹¹

Al año siguiente, en ILSVRC'2014, la tasa de error se redujo drásticamente, del 11.2 % del equipo de Clarifai al 6.66 % de un equipo de Google que bautizó a su red como GoogLeNet en honor a Yann LeCunn. Como en 2013, el vencedor en términos de tasa de error fue un ensemble de seis redes neuronales.

La red de Google reducía drásticamente el número de parámetros de la red. De los 60 millones de parámetros de AlexNet en 2012, se pasaba a “sólo” 4 millones de parámetros. GoogLeNet sustituía las capas finales, completamente conectadas, por una capa de *average pooling* 7×7 (pooling realizando la media aritmética en lugar de quedarnos con el máximo). Esto ahorra un montón de parámetros que demostraron no ser imprescindibles.

La contribución principal del equipo ganador, no obstante, fue el diseño de un módulo, denominado *Inception*, en el que se utilizan

Una red con unidades ReLU se entrena más rápido que una que utilice neuronas sigmoidales, p.ej tanh.

⁶⁹⁰ Matthew D. Zeiler y Rob Fergus. Visualizing and understanding convolutional networks. *arXiv e-prints*, arXiv:1311.2901, 2013. URL <http://arxiv.org/abs/1311.2901>

⁶⁹¹ Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, y Andrew Rabinovich. Going deeper with convolutions. En *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9, 2015a. ISBN 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298594

convoluciones 1×1 . El objetivo del diseño modular de la red era conseguir redes neuronales verdaderamente escalables, eficientes en el uso de energía y económicas en su consumo de memoria. El diseño resultante de la red se basa en dos ideas:

- Cuando diseñamos una capa convolutiva, no sabemos cuál es el tamaño más adecuado para sus filtros. Un filtro de 3×3 no nos proporciona la misma información que un filtro de 5×5 . De antemano, por desgracia, no sabemos cuál puede ser mejor, como tampoco sabemos cuál es la mejor posición de una capa de pooling en la red. La solución: los usamos todos en paralelo, de forma que un módulo Inception contiene bancos de filtros de distintos tamaños (varias capas convolutivas en paralelo), además de una capa de pooling. La salida de un módulo Inception concatena las salidas individuales de las capas de convolución y de pooling que contiene. La siguiente capa de la red decidirá qué parte de su entrada utiliza y cómo lo hace. De esta forma, el propio modelo es capaz de seleccionar qué tipo de capa (convolutiva o pooling) es mejor para cada nivel de la red convolutiva.
- Al poner múltiples capas en paralelo en cada nivel de profundidad, si lo hacemos de forma ingenua, estamos aumentando los requisitos computacionales de la red. Un aumento lineal en el número de filtros convolutivos se traduce en un aumento cuadrático del coste computacional. Si estamos triplicando el número de filtros al utilizar, en paralelo, distintos tamaños de filtro, el coste total de la red se dispara. La solución de Inception: utilizar convoluciones 1×1 para reducir la dimensionalidad de los datos, tanto limitando la profundidad de la salida de cada módulo como la profundidad de las entradas de cada capa convolutiva. Este uso de convoluciones 1×1 se denomina, en ocasiones, capa “*network in network*”, como si tuviésemos una red dentro de la red.

Con este diseño modular, que difiere de la tradicional estructura secuencial de las redes convolutivas tradicionales, se consiguen redes que son, a la vez, profundas (con muchas capas) y anchas (con muchas operaciones en paralelo), por lo que resultan muy flexibles. La primera versión de Inception, GoogLeNet, ganó la competición ILSVRC’2014 usando una red con 22 capas de profundidad (9 módulos Inception y más de 100 capas en total). Versiones posteriores de Inception refactorizaron las convoluciones de mayor tamaño en secuencias de convoluciones más pequeñas, que resultan más fáciles de entrenar. Por ejemplo, en Inception v3,⁶⁹² una convolución 5×5 se sustituye por dos convoluciones 3×3 .

Para terminar, una curiosidad: ¿por qué se llama Inception? Por el meme “we need to go deeper”, de la película del mismo nombre

⁶⁹² Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, y Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *arXiv e-prints*, arXiv:1512.00567, 2015b. URL <http://arxiv.org/abs/1512.00567>

dirigida por Christopher Nolan en 2010.

- *VGGNet*⁶⁹³

El segundo puesto de ILSVRC'2014, en términos de error de clasificación, fue para otra red convolutiva, diseñada por Karen Simonyan y Andrew Zisserman del Visual Geometry Group de la Universidad de Oxford en Inglaterra, por lo que se denomina VGGNet. Su tasa de error, del 7.33 % no estaba tan lejos de los resultados obtenidos por GoogLeNet.

La idea principal que guio su diseño es la simplicidad y la identificación de la profundidad de la red como un factor crítico en su rendimiento. Esa profundidad es la que permite que las redes convolutivas profundas sean capaces de extraer una representación jerárquica de los datos visuales con los que trabajan.

A diferencia de GoogLeNet, la arquitectura de VGGNet es simple y elegante. Sólo realiza convoluciones 3×3 y pooling 2×2 desde su entrada hasta llegar a las capas finales, completamente conectadas. No obstante, requiere muchos más parámetros que GoogLeNet, del orden de 138 millones, por lo que resulta más costosa computacionalmente. En realidad, casi todos sus parámetros corresponden a sus tres capas finales, completamente conectadas (sólo la primera capa completamente conectada contiene más de 100 millones de los 138 millones de parámetros de la red). El coste computacional de la red, como es habitual en las redes convolutivas, viene marcado por sus capas convolutivas.

Observe la evolución en el diseño de redes convolutivas. De los filtros 11×11 de AlexNet en 2012, se pasó a los filtros 7×7 de ZFNet en 2013. De ahí, a utilizar sólo filtros 3×3 en VGGNet (2014). La misma progresión se observa también en las diferentes versiones de Inception. Concatenando varios filtros pequeños se simula el funcionamiento de filtros más grandes manteniendo las ventajas computacionales de los pequeños. Se reduce el número de parámetros de la red y, además, añadiendo unidades ReLU a la salida de las capas convolutivas, se potencia el comportamiento no lineal de la red convolutiva.

- *ResNet*⁶⁹⁴

2015 marcó un hito en la historia de la visión artificial. Por primera vez, un sistema de aprendizaje automático rebajaba la tasa de error de un humano a la hora de clasificar imágenes (estimada en un 5.1 % para ImageNet). Investigadores de Microsoft Research conseguían un error del 4.94 % utilizando unidades PReLU [*Parametric Rectified Linear Unit*],⁶⁹⁵ unidades ReLU parametrizadas que permiten ajustar mejor los parámetros de una red sin apenas coste computacional extra. La diferencia puede parecer pequeña, pero el error de las redes PReLU (o PReLU-nets) supone una mejora relativa del 26 % con respecto al

⁶⁹³ Karen Simonyan y Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR'2015*, arXiv:1409.1556, 2014. URL <http://arxiv.org/abs/1409.1556>

Las capas completamente conectadas contienen más parámetros que las convolutivas, por lo que requieren mayor espacio de almacenamiento en memoria. Sin embargo, es el procesamiento de datos realizado en las capas convolutivas el que requiere más tiempo de “CPU” (a decir verdad, una CPU resulta demasiado lenta en la práctica, por lo que se recurre al uso de una GPU).

⁶⁹⁴ Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. Deep residual learning for image recognition. En *CVPR'2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. DOI: 10.1109/CVPR.2016.90

⁶⁹⁵ Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv e-prints*, 2015a. URL <http://arxiv.org/abs/1502.01852>

ganador del año anterior (GoogLeNet, 6.66 %).

En la competición ILSVRC de ese año, un ensemble de redes entrenadas por el equipo de Microsoft Research rebajó la tasa de error hasta el 3.57 %, casi la mitad que el año anterior y mucho menor que la tasa de error de una persona etiquetando las imágenes manualmente (por no mencionar que el ordenador no se cansa y puede clasificar montones de imágenes por segundo).

Mientras que la idea tras el diseño de GoogLeNet era hacer la red más ancha, la idea de los investigadores de Microsoft era conseguir redes más profundas. El problema es que, cuantas más capas ocultas añadimos a una red, peor suele funcionar. Para solventar este problema, los investigadores de Microsoft crearon las redes de residuos o ResNets [*residual networks*], con las que ganaron la competición usando redes ultra-profundas de 152 capas.

¿Por qué funcionan peor las redes neuronales cuantas más capas añadimos? En teoría, deberían funcionar también como las redes menos profundas, al menos en el conjunto de entrenamiento. Sin embargo, su rendimiento se va deteriorando porque el entrenamiento de cada capa resulta más difícil.

La clave de las ResNets es similar a la codificación diferencial en procesamiento digital de señales. Si resulta complejo aprender la relación entre la entrada x de una capa y su salida $H(x)$, tal vez sea más sencillo aprender su diferencia $H(x) - x$ (su residuo). Luego, sólo tenemos que añadir el residuo a la entrada x para obtener la salida $H(x)$. Si definimos el residuo como $F(x) = H(x) - x$, en lugar de tratar de aprender $H(x)$, construimos una red que intente aprender $F(x) + x$.

Intuitivamente, ¿por qué funcionan tan bien las ResNets? Imaginemos que tenemos una red con n capas. En principio, podríamos construir una red con $n + 1$ capas que funcionase igual de bien y que se limitase, en su primera capa, a copiar la señal de entrada y pasársela a la red de n capas. El problema es que, en una red convencional, resulta difícil entrenar esa primera capa y conseguir que implemente la función identidad. En una ResNet, sin embargo, es mucho más sencillo aprender esta función: basta con ir eliminando el residuo $F(x)$ hasta que sea cero y acabaríamos obteniendo la función identidad como salida del bloque de la ResNet. Esto explica por qué, al usar redes de residuos, no se degrada su rendimiento cuantas más capas utilizamos. Los bloques con los que se construyen las ResNets proporcionan, de entrada, un punto de referencia (la entrada x) a partir del cual podemos entrenar con mayor facilidad redes muy profundas.

Como resultado, en las ResNets aparecen conexiones que se saltan capas [*skip connections*], las correspondientes a la propagación de la entrada que luego se combina con el residuo. De esta forma se facilita

Año	Red	Capas	Error
2012	AlexNet	8	15.32 %
2013	ZFNet	8	11.20 %
2014	VGGNet	19	7.33 %
2014	GoogLeNet	22	6.66 %
2015	PReLU-net	22	4.94 %
2015	ResNet	152	3.57 %

la propagación del gradiente del error a través de la red. Usando esta estrategia se ha conseguido entrenar redes de más de mil capas (que podemos utilizar directamente, ya que Microsoft pone sus redes ya entrenadas a nuestra disposición). Como en GoogLeNet, en las ResNets también se prescinde de las capas completamente conectadas finales y, como en otros modelos de redes convolutivas, es habitual recurrir a técnicas como la normalización por lotes.

Mientras que en las redes multicapa convencionales siempre corremos el riesgo de que el gradiente del error se desvanezca exponencialmente, en una ResNet el gradiente puede propagarse directamente hasta las primeras capas de la red, a través de los atajos proporcionados por las conexiones que propagan la entrada de cada bloque hasta su salida para combinarla con los residuos.

El éxito de las redes de residuos ha sido tal, que versiones revisadas de Inception (la arquitectura de Google) incluyen conexiones residuales en cada módulo, por lo que Inception v4 es, realmente, un híbrido Inception-ResNet.⁶⁹⁶

En definitiva, el progreso experimentado en este campo ha hecho que, si disponemos de miles de ejemplos por clase, una red neuronal artificial sea capaz de entrenarse con éxito para reconocer los objetos principales que aparecen en una imagen y clasificar correctamente la imagen de acuerdo al conjunto de clases que se haya definido (que puede corresponder a marcas de vehículos, especies de plantas, razas de perros o cualquier otra clasificación que sea de interés).

Hay que mencionar, no obstante, que aunque la tasa de error de una red convolutiva pueda ser menor que la tasa de error de un humano, el tipo de errores que cometen suele ser diferente. Por ejemplo, la taxonomía definida por WordNet y, por extensión, ImageNet, incluye 120 razas de perros. Un ser humano puede que no sea capaz de identificar correctamente la raza de un perro (tal vez no conozca la diferencia entre un labrador y un golden retriever) y se contabilice como error que fue incapaz de llegar a ese grado de detalle, que una red bien entrenada probablemente sí sea capaz de distinguir. En cambio, una red convolutiva puede que confunda un cepillo de dientes con un bate de béisbol, algo que ni siquiera un niño pequeño haría.

Tabla 8: La evolución de las redes convolutivas a la hora de clasificar imágenes de ImageNet. La tasa de error corresponde a la frecuencia con la que la red no es capaz de identificar la clase correcta entre sus 5 primeras respuestas [*top-5 classification*]. Como referencia, la tasa de error de un ser humano realizando la misma tarea es del 5.1 %

⁶⁹⁶ Christian Szegedy, Sergey Ioffe, y Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv e-prints*, arXiv:1602.07261, 2016. URL <http://arxiv.org/abs/1602.07261>

En las redes que utilizan capas completamente conectadas tras las capas convolutivas, la mayoría de sus parámetros corresponden a dichas capas. Es el caso de AlexNet, con el 96 % de sus 61 millones de parámetros en sus capas FC, o VGG-19, con el 90 % de sus 138M parámetros en sus capas FC. Esto hace que el simple hecho de almacenar la red ya suponga un coste significativo en memoria: sobre 240MB en AlexNet, 552MB en VGG-19. En modelos posteriores, como Inception o ResNets, estas capas desaparecen y se sustituyen por capas de pooling (*average pooling*, para ser precisos). Esto puede suponer un ahorro considerable con respecto al tamaño de la red en cuanto a su número de parámetros, aunque no necesariamente con respecto a su uso de memoria en tiempo de ejecución, que depende principalmente de las dimensiones de sus capas convolutivas. Por no hablar del coste computacional que supone su uso: AlexNet “sólo” necesitaba 0.72GFLOPS para clasificar una imagen, frente a los 1.5GFLOPS de GoogLeNet, 11.3GFLOPS de ResNet-152 o 15.5GFLOPS de VGG-16.

Conforme las tasas de error se han ido reduciendo sistemáticamente, se han investigado formas de optimizar los recursos necesarios para un modelo de clasificación ya entrenado, como su número de parámetros (que se cuentan en millones), sus requisitos de memoria (en MB), el número de operaciones aritméticas que requiere su utilización (en TFLOPS), su tiempo de respuesta a la hora de realizar inferencias (predicciones) y su consumo energético.⁶⁹⁷ Estas son sólo algunas de las estrategias que se pueden utilizar:

- *Poda [pruning]*: Eliminar todas aquellas conexiones cuyos pesos, en términos absolutos queden por debajo de un umbral. Es la técnica de compresión más sencilla y rápida, que además se puede conseguir sin pérdidas en la precisión del modelo. Por ejemplo, se puede lograr una compresión 9x de AlexNet o 13x de VGG-16.
- *Pesos compartidos*: Agrupar todos los pesos que tienen valores similares y almacenarlos en un diccionario, usando un libro de código [*codebook*], codificación Huffman o HashedNets.⁶⁹⁸ Por ejemplo, si redondeamos los valores de los pesos de forma que sólo tomen 256 valores diferentes, podemos reducir el tamaño del fichero necesario en TensorFlow para representar Inception/GoogLeNet de 87MB a 26MB, con sólo un 1 % de pérdida en la precisión de la red.
- *Redes poco profundas [shallow networks]*: En lugar de entrenar una red muy profunda, se puede utilizar una red menos profunda y entrenarla con un conjunto de datos aumentado que se limite a reproducir las condiciones bajo las cuales la red se va a utilizar (p.ej. GroupLens o Google Translate).
- *Filtros pequeños*: Un filtro de convolución de tamaño 7×7 se puede sustituir por una secuencia de 3 convoluciones 3×3 . Esta sustitución

⁶⁹⁷ Alfredo Canziani, Adam Paszke, y Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv e-prints*, arXiv:1605.07678, 2016. URL <http://arxiv.org/abs/1605.07678>

⁶⁹⁸ Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, y Yixin Chen. Compressing neural networks with the hashing trick. *arXiv e-prints*, arXiv:1504.04788, 2015b. URL <http://arxiv.org/abs/1504.04788>

hace que la red tenga menos parámetros, requiera menos potencia de cálculo y, además, incorpore un grado mayor de no linealidad (añadiendo rectificadores ReLU en las salidas de las convoluciones intermedias).

- *Precisión reducida:* El simple hecho de realizar operaciones en coma flotante con menos bits de precisión (p.ej. de 32 a 16 bits) puede suponer un ahorro notable en términos de uso de energía y no afectar demasiado a la precisión de la red. Es una tendencia que se observa en el diseño arquitectónico de las nuevas generaciones de GPU. Las NVidia Pascal incluyen núcleos para realizar operaciones en coma flotante de 32 (FP32) y 64 bits (FP64), con el doble de núcleos de 32 que de 64 bits, así como núcleos para enteros de 8 bits (INT8). La siguiente generación de GPUs, NVidia Volta, incluye núcleos de precisión mixta de 16 y 32 bits (FP16/FP32).
- *Pesos binarios:* Llevada al extremo, la reducción de precisión puede también aplicarse a los pesos de la red, hasta el punto de hacer que todos los pesos sean binarios (más concretamente, bipolares: +1 ó -1). Con pesos binarios, la operación de convolución se convierte en una sucesión de sumas y restas que puede realizarse de forma mucho más rápida que las operaciones en coma flotante. Es el caso de XNOR-Net.⁶⁹⁹

Combinando una o varias de las estrategias anteriores, se pueden conseguir redes convolutivas lo suficientemente pequeñas en cuanto a sus necesidades computacionales como para que puedan ejecutarse directamente en un dispositivo móvil, como un smartphone o una tablet. Por ejemplo, SqueezeNet consigue la precisión de AlexNet (80.5 % de precisión top-5 en ImageNet), pero con 50 veces menos parámetros que AlexNet y ocupando menos de 0.5MB de memoria (500 veces menos que AlexNet).⁷⁰⁰

Las anteriores son sólo algunas de las vías que se han explorado para mejorar, en distintos aspectos, el diseño de redes convolutivas. Otras alternativas interesantes, de las muchas que se han propuesto, incluyen las siguientes:

- *DenseNets*⁷⁰¹

Las redes convolutivas densamente conectadas introducen conexiones directas entre las salidas de una capa convolutiva y las entradas de todas las capas convolutivas siguientes, con la misma intención que los atajos de las ResNets. Mientras que una red convolutiva convencional de L capas tiene L conexiones entre capas, una DenseNet tiene $L(L + 1)/2$ conexiones directas. De esta forma, eliminan de un plumazo el problema del gradiente evanescente [*vanishing gradient*] en el entrenamiento de redes.

⁶⁹⁹ Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, y Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *arXiv*, arXiv:1603.05279, 2016b. URL <http://arxiv.org/abs/1603.05279>

⁷⁰⁰ Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, y Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *arXiv e-prints*, arXiv:1602.07360, 2016. URL <http://arxiv.org/abs/1602.07360>

⁷⁰¹ Gao Huang, Zhuang Liu, y Kilian Q. Weinberger. Densely connected convolutional networks. *arXiv e-prints*, arXiv:1608.06993, 2016. URL <http://arxiv.org/abs/1608.06993>

La misma idea la propusieron, años antes, investigadores de Microsoft Research para diseñar redes neuronales con las que mejorar el rendimiento de los sistemas de reconocimiento de voz, otro de los grandes éxitos de las redes neuronales (no hay sistema de reconocimiento de voz actual que no incorpore, internamente, una red neuronal). Esas redes, conocidas como redes apiladas profundas [*deep stacking networks*], se conocían originalmente como redes convexas profundas [*deep convex networks*].⁷⁰²

- *Convoluciones dilatadas [dilated convolutions]*⁷⁰³

Las redes con convoluciones dilatadas añaden un nuevo hiperparámetro a las redes convolutivas: su dilatación. En lugar de emplear convoluciones que se aplican sobre posiciones contiguas de la imagen, la dilatación permite que existan espacios entre las entradas de una operación de convolución. Algo así como definir un tamaño de paso para los elementos que intervienen en la convolución igual que se define un paso [*stride*] entre convoluciones consecutivas.

Por ejemplo, si no usamos dilatación, $D = 0$, calcularíamos la convolución de la forma tradicional: $\sum w_i x_i$. Si introducimos dilatación, $D = 1$, la operación de convolución sería de la forma $\sum w_i x_{2i}$, insertando un hueco entre cada pareja de entradas consecutivas que intervienen en la convolución.

Esto nos puede permitir combinar información espacial de forma mucho más agresiva utilizando menos capas convolutivas en la red. Si apilamos k capas convolutivas convencionales de tamaño 3×3 , cubriremos un espacio de tamaño $(1 + 2k) \times (1 + 2k)$. Si apilamos capas convolutivas dilatadas, podemos conseguir que el espacio cubierto vaya creciendo exponencialmente (aumentando exponencialmente el tamaño de la dilatación).

- *Xception*⁷⁰⁴

Xception, una red diseñada por François Chollet, el autor de Keras, lleva al extremo los principios de Inception, de ahí su nombre: Xception como síncopa de “*extreme inception*”. Se basa en la idea de que las correlaciones entre canales y las correlaciones espaciales están lo suficientemente desacopladas como para que sea preferible hacerlas por separado.

En una capa convolutiva estándar, el filtro de convolución se aplica espacialmente y también entre canales (tiene en cuenta la profundidad de su entrada). En Inception, los canales se separan (o, más bien, se mezclan) utilizando convoluciones 1×1 para reducir la dimensionalidad de los datos antes de realizar la convolución espacial. En Xception, se va un paso más allá: las convoluciones espaciales se realizan de forma independiente para cada canal y el resultado se combinan utilizando una convolución 1×1 para capturar las interacciones entre distintos

⁷⁰² Dong Yu y Li Deng. Deep convex net: A scalable architecture for speech pattern classification. En *INTERSPEECH 2011, 12th Annual Conference of the International Speech Communication Association, Florence, Italy, August 27-31, 2011*, pages 2285–2288. ISCA, 2011. URL <https://goo.gl/cbvEi9>

⁷⁰³ Fisher Yu y Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *ICLR'2016*, arXiv:1511.07122, 2015. URL <http://arxiv.org/abs/1511.07122>

⁷⁰⁴ François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv e-prints*, arXiv:1610.02357, 2016. URL <http://arxiv.org/abs/1610.02357>

canales.

El proceso es equivalente a sustituir la convolución $k \times k \times d$ por dos convoluciones separadas: una espacial $k \times k$ aplicada de forma independiente para cada canal y otra en profundidad, 1×1 , entre canales. Se sustituye el aprendizaje de las máscaras de convolución 3D de una capa convolutiva convencional por el aprendizaje de un conjunto de máscaras 2D seguido de una máscara 1D, lo que en principio puede facilitar el aprendizaje. ¿Lo hace? Al menos en ImageNet, Xception mejora ligeramente los resultados obtenidos por Inception v3, resultando más eficiente.

MobileNet, una versión de Xception optimizada para dispositivos móviles, está detrás de algunas aplicaciones de Google para dispositivos móviles.⁷⁰⁵

■ *Redes de transformación espacial STN*⁷⁰⁶

Las redes STN [*Spatial Transformer Networks*], propuestas por un equipo de Google DeepMind, introducen un módulo en las redes convolutivas cuyo objetivo es conseguir que a las capas posteriores de la red les resulte más sencillo clasificar correctamente. En vez de manipular la estructura en sí de las capas de la red convolutiva, manipulan la imagen antes de que se le suministre a una capa convolutiva.

El módulo de transformación espacial pretende normalizar la posición del objeto de interés (corregir traslaciones, rotaciones y cambios de escala). En lugar de muestrear espacialmente de forma uniforme, como se hace con *max pooling*, cuya misión es detectar una característica específica en una región dada olvidándose de su posición exacta, el módulo STN es dinámico en el sentido de que actúa de distinta forma para cada imagen de entrada; esto es, la distorsiona/transforma de diferentes maneras.

Un módulo STN consiste en una red de “localización” y un “muestreador” [*sampler*]. La red de localización genera los parámetros de la transformación espacial que debe aplicarse, representada mediante un tensor 6D para transformaciones afines. El *sampler* aplica esa transformación sobre la entrada, deformándola de acuerdo a los parámetros suministrados por la red de localización.

En una red convolutiva, se puede insertar un módulo STN en cualquier punto de su estructura. Básicamente, el módulo ayuda a la red a aprender cómo transformar los mapas de características de forma que se minimice mejor la función de coste durante el entrenamiento de la red. Como vemos, no siempre se mejoran los resultados modificando el diseño interno de los módulos de una red, como sucedió con Inception o ResNet. En ocasiones, basta con añadir el componente adecuado en el lugar adecuado, dentro de una arquitectura de red convencional.

⁷⁰⁵ Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, WeiJun Wang, Tobias Weyand, Marco Andreetto, y Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv e-prints*, arXiv:1704.04861, 2017. URL <http://arxiv.org/abs/1704.04861>

⁷⁰⁶ Max Jaderberg, Karen Simonyan, Andrew Zisserman, y Koray Kavukcuoglu. Spatial transformer networks. *arXiv e-prints*, arXiv:1506.02025, 2015. URL <http://arxiv.org/abs/1506.02025>

Como vemos, el abanico de posibilidades es muy extenso y nunca sabemos qué estrategia heurística puede dar el mejor resultado en el futuro. Por no hablar de las posibles extensiones de todos estos modelos para trabajar con señales de vídeo, que no son más que secuencias de imágenes en movimiento,⁷⁰⁷ o imágenes tridimensionales, en las que los píxeles se sustituyen por véxoles y las convoluciones en el plano pasan a ser convoluciones volumétricas, dando lugar a las redes convolutivas volumétricas [*Volumetric CNNs*].⁷⁰⁸

Aunque los resultados conseguidos por la redes convolutivas sean sorprendentes, si los comparamos con lo que se podía hacer hasta hace sólo unos años, aún queda mucho trabajo por delante... Al clasificar vídeos de YouTube del conjunto de datos Kinetics,⁷⁰⁹ las redes convolutivas pueden conseguir una precisión superior al 80% cuando se trata de identificar algunas tareas como jugar al tenis. Sin embargo, bajan al 20% (o menos) cuando intentan clasificar las acciones favoritas de Homer Simpson, como beber cerveza o comer donuts.

Detección de objetos

La clasificación de imágenes no es el único problema de visión artificial en el que se han utilizado con éxito las redes neuronales convolutivas. Otro problema relacionado es la detección de objetos. En este problema, tenemos imágenes en las que pueden aparecer uno o varios objetos que deseamos identificar y localizar dentro de la imagen. El problema es mucho más complejo que la simple clasificación de imágenes porque, a priori, desconocemos el número de objetos relevantes que aparecen en la imagen, su tamaño y su posición. Además, algunos objetos pueden aparecer parcialmente ocluidos por otros objetos que estén delante.

La primera solución que a uno se le puede ocurrir consiste en utilizar una ventana deslizante que iremos desplazando sobre la imagen original. Para cada posición en la que coloquemos la ventana, extraemos una región rectangular de la imagen original y utilizamos un clasificador de imágenes para detectar si en esa región aparece alguno de los objetos que sean de interés para nosotros. Es una solución conceptualmente sencilla, fácil de implementar, pero no siempre eficiente.

Antes de que se empleasen redes convolutivas para resolver este problema, ya existían algunas técnicas que se podían emplear en dominios de aplicación concretos. Basten un par de ejemplos:

- La solución clásica de Paul Viola y Michael Jones,⁷¹⁰ de Compaq (empresa de ordenadores adquirida por Hewlett-Packard en 2002), se basa en utilizar miles de clasificadores basados en características Haar extraídas directamente de la imagen de forma eficiente, lo que permite su uso en tiempo real. El algoritmo de Viola y Jones se emplea con éxito en la detección de caras en imágenes⁷¹¹ y es la base de los

⁷⁰⁷ Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, y Li Fei-Fei. Large-scale video classification with convolutional neural networks. En *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014. DOI: 10.1109/CVPR.2014.223

⁷⁰⁸ Vishakh Hegde y Reza Zadeh. FusionNet: 3D Object Classification Using Multiple Data Representations. *arXiv e-prints*, arXiv:1607.05695, 2016. URL <http://arxiv.org/abs/1607.05695>

⁷⁰⁹ Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, y Andrew Zisserman. The kinetics human action video dataset. *arXiv e-prints*, arXiv:1705.06950, 2017. URL <http://arxiv.org/abs/1705.06950>

⁷¹⁰ Paul Viola y Michael J. Jones. Robust real-time object detection. Technical Report CLR 2001/01, Cambridge Research Laboratory, Compaq, February 2001. URL <http://www.hpl.hp.com/techreports/Compaq-DEC/CRL-2001-1.pdf>

⁷¹¹ Paul Viola y Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004. ISSN 1573-1405. DOI: 10.1023/B:VISI.0000013087.49260.fb

Conjunto de datos	Categorías	Imágenes	URL
ImageNet	200	450k	http://www.image-net.org/
COCO	80	120k	http://cocodataset.org/
Pascal VOC	20	12k	http://host.robots.ox.ac.uk/pascal/VOC/
Oxford-IIIT Pet	37	7k	http://www.robots.ox.ac.uk/~vgg/data/pets/
KITTI Vision	3	7k	http://www.cvlibs.net/datasets/kitti/

algoritmos utilizados por multitud de cámaras fotográficas digitales.

- Otro método basado en realizar ingeniería de características, en este caso histogramas de gradientes orientados [*HOG, Histogram of Oriented Gradients*], se puede emplear para localizar figuras humanas en imágenes. Con esta técnica se pueden detectar peatones, generalmente en posición vertical, aunque bajo diferentes poses, condiciones de iluminación, apariencia y ropa, sobre distintos fondos y con occlusiones parciales que complican la detección.⁷¹²

Dado el éxito obtenido por las redes neuronales convolutivas en la clasificación de imágenes, no tardaron en aparecer soluciones para la detección de objetos basadas en el uso de redes convolutivas:

- *OverFeat*⁷¹³

Las primeras soluciones basadas en *deep learning*, como OverFeat,⁷¹⁴ utilizaban ventanas deslizantes multiescala y redes convolutivas de clasificación de imágenes. OverFeat, diseñada por el equipo de Yann LeCun en la Universidad de Nueva York, se proclamó vencedor en el apartado de localización de objetos en la competición ILSVRC'2013, igual que hiciese un año antes AlexNet en clasificación de imágenes.

- *Redes convolutivas basadas en regiones [R-CNN]*⁷¹⁵

Al año siguiente, en 2014, un equipo de la Universidad de California en Berkeley propuso el uso de redes convolutivas basadas en regiones [*R-CNN, Region-Based Convolutional Networks*]. Con ellas, se mejoró casi un 50% la precisión en el problema de detección de objetos.

La propuesta original,⁷¹⁶ descompone el problema en tres etapas. En primer lugar, se extraen posibles objetos utilizando un método de proposición de regiones, que suele consistir en un proceso de búsqueda selectiva. En segundo lugar, para cada región propuesta, se extraen características utilizando una red convolutiva de las entrenadas sobre ImageNet para clasificar imágenes. En una tercera etapa, se clasifica cada región utilizando máquinas lineales de vectores de soporte [*LSVMs: Linear Support Vector Machines*] a partir de las características extraídas de la imagen por la red convolutiva y se emplea un algoritmo de regresión para intentar delimitar mejor las fronteras del

Tabla 9: Algunos conjuntos de datos estándar utilizados en la evaluación de técnicas de detección de objetos.

⁷¹² Navneet Dalal y Bill Triggs. Histograms of oriented gradients for human detection. En *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893, 2005. DOI: 10.1109/CVPR.2005.177

⁷¹³ Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, y Yann LeCun. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. *arXiv e-prints*, arXiv:1312.6229, 2013. URL <http://arxiv.org/abs/1312.6229>

⁷¹⁴ Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, y Yann LeCun. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. *arXiv e-prints*, arXiv:1312.6229, 2013. URL <http://arxiv.org/abs/1312.6229>

⁷¹⁵ Ross B. Girshick, Jeff Donahue, Trevor Darrell, y Jitendra Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(1):142–158, 2016. ISSN 0162-8828. DOI: 10.1109/TPAMI.2015.2437384

⁷¹⁶ Ross B. Girshick, Jeff Donahue, Trevor Darrell, y Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CVPR'2014*, arXiv:1311.2524v5, 2014a. URL <http://arxiv.org/abs/1311.2524>

objeto en la imagen [*bounding box regressor*]. Por último, si varios de los objetos identificados se solapan demasiado, se eliminan los menos probables para reducir el número final de objetos identificados.

En principio, cualquier método de proposición de regiones podría servir. La búsqueda selectiva, utilizada en la versión original de R-CNN, genera 2000 regiones diferentes en las que existe una alta probabilidad de que un objeto de interés esté presente. Es decir, se traduce el problema de detección de objetos dentro de una imagen en un problema de clasificación de 2000 imágenes. Es más eficiente que ir probando con una ventana deslizante a ciegas, pero sigue siendo computacionalmente costoso (casi un minuto por cada imagen procesada).

No tardaron en surgir nuevas variantes que intentaban acelerar el proceso de detección de imágenes, como Fast R-CNN o Faster R-CNN.

- *Fast R-CNN*⁷¹⁷

Ya trabajando para Microsoft Research, Ross Girshick mejoró la propuesta original sustituyendo el uso final de clasificadores SVM por una solución pura de *deep learning*. Una red multicapa final, de tipo convencional, que se encarga tanto de clasificar la región de la imagen como de hacer una regresión para delimitar los límites del objeto identificado en esa región. El modelo resultante es más sencillo de entrenar (una red neuronal, perfectamente diferenciable), aunque se basa aún en el uso de búsqueda selectiva (o cualquier otro método de proposición de regiones).

En Fast R-CNN, la red convolutiva se aplica sobre la imagen completa y se utiliza un mecanismo de pooling RoI [*region of interest pooling*] para ir seleccionando regiones de interés. Es decir, se intercambia el orden con respecto a R-CNN, donde primero se seleccionaban las regiones y luego se aplicaba la red convolutiva. Esto supone un ahorro computacional (sólo se recurre a la red convolutiva una vez, para analizar toda la imagen de golpe), aunque el proceso de inferencia sigue siendo costoso computacionalmente (hay que ir probando con todas las regiones propuestas).

- *Faster R-CNN*⁷¹⁸

En colaboración con Shaoqing Ren, Ross Girshick diseñó una tercera generación de R-CNN, denominada Faster R-CNN. Este modelo consigue eliminar el algoritmo de búsqueda selectiva que iba generando regiones de interés y lo sustituye por una red de proposición de regiones RPN [*region proposal network*], que se inserta tras la última capa convolutiva de la red y es capaz de generar propuestas de regiones que luego se usan en el resto de la red como en Fast R-CNN (RoI pooling, capas completamente conectadas, clasificación y regresión). Con el uso de este módulo, la red RPN, se consigue que la detección de objetos se pueda realizar

⁷¹⁷ Ross B. Girshick. Fast r-cnn. En *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015a. DOI: 10.1109/ICCV.2015.169

⁷¹⁸ Shaoqing Ren, Kaiming He, Ross B. Girshick, y Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *arXiv e-prints*, arXiv:1506.01497v2, 2016. URL <http://arxiv.org/abs/1506.01497>

en tiempo real.

- *R-FCN [Region-based Fully Convolutional Network]*⁷¹⁹

También en Microsoft Research, se diseñó una red completamente convolutiva para detectar objetos: R-FCN. A diferencia de Faster R-CNN y sus versiones anteriores, que realizan operaciones costosas por cada región de interés, en R-FCN se comparten casi todos los cálculos para toda la imagen. Además, este modelo puede adoptar, como columna vertebral, una red convolutiva entrenada para clasificar imágenes (como ResNet, por ejemplo), con lo que logra resultados muy competitivos y, además, sólo necesita milésimas de segundo para procesar cada imagen completa (de 2.5 a 20 veces más rápido que Faster R-CNN).

- *Mask R-CNN*⁷²⁰

Ross Girshick, tras dejar Microsoft Research y comenzar a trabajar para Facebook en FAIR, su laboratorio de IA, extendió Faster R-CNN de una forma distinta. A la arquitectura de Faster R-CNN le añadió una rama para predecir una máscara para el objeto en paralelo a la identificación de su región en la imagen. Mask R-CNN es ligeramente más costoso que Faster R-CNN, pero se puede seguir utilizando en tiempo real (a 5 fps, frames por segundo). Su diseño le permite conseguir resultados excepcionales en la resolución de tareas como localizar objetos, segmentar imágenes o determinar la pose de una persona (identificando puntos clave de su anatomía).

Las redes convolutivas basadas en regiones no son las únicas que se han utilizado con éxito en la detección de objetos. También destaca un algoritmo que recoge algunas ideas interesantes. Un algoritmo con un nombre peculiar...

- *YOLO [You Only Look Once]*^{721,722}

El algoritmo YOLO proporciona una estrategia alternativa para detectar objetos en tiempo real, sin necesidad de utilizar regiones. Básicamente, YOLO convierte el problema de detección de objetos en un problema de regresión que se resuelve con una red convolutiva única. Como su nombre indica, sólo se mira una vez la imagen.

El algoritmo YOLO parte de una imagen, que dividimos en un conjunto de celdas (p.ej. dibujando sobre ella una rejilla 19×19). Entonces, una red neuronal, para cada celda de la rejilla en la que hayamos dividido la imagen, genera un vector con las probabilidades de que esa celda contenga un objeto de cada una de las clases de nuestro problema. Además, crea simultáneamente un modelo de regresión para delimitar las regiones [*anchor boxes*] que contienen los objetos de cada celda. Una vez que, para cada celda, obtenemos una o varias regiones bien delimitadas, eliminamos aquéllas en las que sea poco probable que exista un objeto (cuando la predicción indica una probabilidad

⁷¹⁹ Jifeng Dai, Yi Li, Kaiming He, y Jian Sun. R-FCN: object detection via region-based fully convolutional networks. *arXiv e-prints*, arXiv:1605.06409, 2016. URL <http://arxiv.org/abs/1605.06409>

⁷²⁰ Kaiming He, Georgia Gkioxari, Piotr Dollár, y Ross B. Girshick. Mask R-CNN. *arXiv e-prints*, arXiv:1703.06870, 2017. URL <http://arxiv.org/abs/1703.06870>

⁷²¹ Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, y Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *arXiv e-prints*, arXiv:1506.02640, 2015. URL <http://arxiv.org/abs/1506.02640>

⁷²² Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, y Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. En *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 779–788, 2016. ISBN 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.91

baja para todas las clases de nuestro interés). Por último, suprimimos los solapamientos entre las regiones propuestas, igual que en R-CNN [*non-max suppression*]. Es importante que este paso lo hagamos de forma independiente para cada clase, ya que pueden aparecer objetos solapados de diferentes clases en la imagen original.

Como sucedía con las técnicas basadas en el uso de regiones, el progreso en el área es continuo y se han propuesto distintas variantes de YOLO para mejorar su precisión y/o su eficiencia. Entre ellas se encuentran SSD [*Single Shot Detector*]⁷²³ o YOLO'9000 (YOLO v2).⁷²⁴

Los algoritmos de detección de objetos basados en redes convolutivas consiguen resultados espectaculares, siendo capaces de identificar múltiples objetos solapados en imágenes reales, por lo que han supuesto un salto cualitativo notable dentro del campo de la visión artificial.

Reconocimiento facial

El rendimiento de las redes neuronales, a día de hoy, supera al de otras técnicas conocidas cuando disponemos de grandes conjuntos de datos. De hecho, se suele decir que, a diferencia de otras técnicas de aprendizaje automático, el *deep learning* funciona mejor cuantos más datos tengamos. Esta característica es su principal virtud y, simultáneamente, su talón de Aquiles. En muchas ocasiones no disponemos de demasiados datos y, pese a ello, nos gustaría ser capaces de aprender.

En el caso extremo, conocido como [*one-shot learning*], sólo disponemos de un único ejemplo por clase. Que sea extremo, no quiere decir que sea poco habitual. La mayoría de los sistemas de reconocimiento facial, así como otros sistemas biométricos, deben ser capaces de reconocer correctamente a una persona disponiendo de un único ejemplo. En el caso de los sistemas de reconocimiento facial, sólo tenemos una imagen digitalizada de una fotografía de esa persona.

En términos de problemas de clasificación, nuestro conjunto de entrenamiento contiene un ejemplo por clase (y la posibilidad de que la persona que intente acceder al edificio no sea ninguna de las personas autorizadas). No nos valdrá, por tanto construir un clasificador convencional basado en una red convolutiva y una capa softmax. Aparte de que resultaría difícil entrenar un clasificador robusto en estas circunstancias, tendríamos el inconveniente añadido de tener que entrenar por completo el modelo cada vez que diésemos de alta o de baja a alguna persona (cambiando el número de salidas de la capa softmax).

Una forma de resolver este tipo de problemas es, en vez de aprender un clasificador que nos proporcione una distribución de probabilidad sobre las clases del problema, aprender una función de similitud. Esta función de similitud la utilizaremos para comparar parejas de imágenes y determinar

⁷²³ Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, y Alexander C. Berg. SSD: single shot multibox detector. *arXiv e-prints*, arXiv:1512.02325, 2015. URL <http://arxiv.org/abs/1512.02325>

⁷²⁴ Joseph Redmon y Ali Farhadi. YOLO9000: better, faster, stronger. En *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6517–6525, 2017. ISBN 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.690

su grado de parecido. O, de forma equivalente, sus diferencias, por lo que utilizaremos una función de distancia d que nos mida la disimilitud entre dos imágenes. Si las imágenes son de la misma persona, la función nos debería devolver un valor bajo. Si son de diferentes personas, su distancia será superior a un umbral que utilizaremos como criterio para tomar nuestra decisión de clasificación. Cuando queramos saber a qué persona corresponde una imagen que nos acaba de llegar, sólo tenemos que comparar esa imagen con las imágenes correspondientes a todas las personas de nuestra base de datos (usando índices y otras técnicas complementarias que nos permitan reducir el número de comprobaciones necesario, si en nuestra base de datos tenemos muchas clases diferentes). Cuando la imagen sea de una persona que no está en nuestra base de datos, ninguna comparación dará una distancia inferior al umbral, lo que nos permitirá tratar con ejemplos de “clases” diferentes a las de nuestro problema de clasificación *one-shot*.

Aunque no sea el campo en el que los algoritmos de *deep learning* se encuentran más cómodos, veamos cómo podríamos resolver problemas de este tipo usando redes neuronales siamesas.

Nuestro objetivo es aprender una función de distancia d que, dados dos ejemplos, nos indique si ambos corresponden a la misma clase o no.

En el caso de las imágenes de un sistema de reconocimiento facial, podemos recurrir a una red convolutiva y quedarnos con la salida de la última capa convolutiva de la red. Esa capa nos proporciona un vector de características que describe, en cierto modo, el contenido de la imagen. En lugar de utilizar ese vector como entrada de una red multicapa convencional que termine en un clasificador softmax, nos vamos a quedar directamente con ese vector de características f .

Si a la red le suministramos de entrada dos ejemplos, x_1 y x_2 , los vectores de características asociados a ambos, $f(x_1)$ y $f(x_2)$, nos pueden servir para aprender la función de distancia d . Las codificaciones de los ejemplos, denominadas *encodings* o *embeddings*, nos proporcionan una forma de medir esa distancia, por ejemplo $d(x_1, x_2) = \|f(x_1) - f(x_2)\|_2^2$.

Dos redes siamesas son, simplemente, dos copias idénticas de una red neuronal que se aplican sobre dos entradas diferentes para comparar sus salidas. ¿Cómo entrenamos redes de este tipo? Como las redes son idénticas, sus parámetros coinciden. Lo único que necesitamos es ajustar esos parámetros, utilizados para calcular los vectores $f(x)$, de forma que la distancia $\|f(x_1) - f(x_2)\|_2^2$ sea pequeña cuando los dos ejemplos corresponden a la misma clase y sea grande cuando pertenezcan a clases diferentes. Conforme vayamos ajustando los parámetros de la red, algo que podemos hacer con técnicas de optimización basadas en el gradiente descendente y *backpropagation*, se va modificando la forma en que se calculan los vectores de características $f(x)$. Al final de su entrenamiento, la red siamesa calculará vectores de características que nos permitan

distinguir parejas de ejemplos de la misma clase de parejas de ejemplos de clases diferentes.

Esta estrategia es la que utiliza Facebook en su sistema de detección facial DeepFace.⁷²⁵ Este sistema tiene una precisión del 97 % frente al 85 % del sistema de identificación utilizado por el FBI.

Es también la estrategia de otro sistema, llamado FaceNet,⁷²⁶ que presume de una precisión récord del 99.63 % sobre el conjunto de datos LFW [*Labeled Faces in the Wild*].

¿Cómo definimos la función objetivo que nos permita entrenar una red siamesa usando el gradiente descendente? Una forma de hacerlo es utilizar una función de pérdida de trillizos [*triplet loss*].

Cuando comparemos parejas de ejemplos, queremos que su codificación f sea similar si corresponden a la misma clase, mientras que queremos que sus vectores de características sean diferentes en caso contrario. Usando la terminología habitual cuando se habla de redes siamesas y trillizos, lo que hacemos es seleccionar un ejemplo particular, a , que usaremos como referencia o ancla [*anchor*]. Por un lado, mediremos la distancia de ese ejemplo con un ejemplo positivo, p , de la misma clase: $d(a, p)$. Además, mediremos la distancia del ejemplo de referencia con respecto a un ejemplo negativo, n , de una clase diferente: $d(a, n)$. Como siempre trabajamos con tres ejemplos a la vez, de ahí proviene el uso del término trillizos.

Nuestro objetivo es conseguir que la distancia $d(a, p)$ sea menor que la distancia $d(a, n)$. Expresado de otra forma, queremos que se verifique la condición

$$d(a, p) - d(a, n) + \alpha \leq 0$$

donde $\alpha > 0$ es un margen que introducimos para evitar que la red aprenda la función trivial $d(x_i, x_j) = 0$ o que use siempre la misma codificación $f(x)$ para todos los ejemplos, situaciones ambas que satisfarían la desigualdad en ausencia de α . De paso, el hiperparámetro α establece un margen que fuerza que existan diferencias entre $d(a, p)$ y $d(a, n)$, separando ambos valores de forma análoga a los márgenes de los clasificadores SVM.

Teniendo lo anterior en cuenta, la función de coste o pérdida con la que tenemos que entrenar la red siamesa es la función de pérdida de trillizos [*triplet loss*], definida sobre tripletas de ejemplos:

$$\mathcal{L}(a, p, n) = \max\{d(a, p) - d(a, n) + \alpha, 0\}$$

Mientras que se satisfaga la desigualdad que nos interesa, la pérdida es cero. Cuando no suceda esto, la pérdida nos indica el error que tenemos que rectificar entrenando nuestra red neuronal. Al minimizar esta función de pérdida, estamos entrenando un modelo con el que resolver problemas de clasificación usando redes neuronales aunque no dispongamos de muchos ejemplos por clase.

⁷²⁵ Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, y Lior Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. En *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 1701–1708, June 2014. ISBN 978-1-4799-5118-5. DOI: 10.1109/CVPR.2014.220

⁷²⁶ Florian Schroff, Dmitry Kalenichenko, y James Philbin. Facenet: A unified embedding for face recognition and clustering. En *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 815–823, 2015. ISBN 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298682

Para entrenar el sistema, siempre harán falta parejas de ejemplos de una misma clase, para poder calcular $d(a, p)$. Pero eso sólo es necesario durante el entrenamiento de la red. No impide que luego apliquemos la red para identificar clases de ejemplos para las que sólo se dispone de un ejemplo [*one-shot learning*]. Ese ejemplo particular no habrá intervenido en el entrenamiento de los parámetros de la red pero, si el modelo aprendido generaliza correctamente, esperamos que ese ejemplo se pueda emplear para clasificar datos correctamente cuando resulte necesario.

En cuanto al conjunto de entrenamiento, ¿cómo se eligen las tripletas? En primer lugar, se forman pares (a, p) de ejemplos de la misma clase. Si escogemos el ejemplo negativo n al azar, es muy fácil conseguir que se satisfaga la restricción de nuestra función de pérdida: es muy probable que el par (a, n) sea muy diferente en comparación con el par a, p . La red no aprenderá mucho escogiendo los ejemplos negativos aleatoriamente. Lo ideal es escoger tripletas (a, p, n) que resulten difíciles de diferenciar, buscando ejemplos en los que $d(a, n)$ sea similar a $d(a, p)$. De esta forma, forzamos al algoritmo de entrenamiento de la red para que extraiga características verdaderamente útiles a la hora de discriminar si una pareja de ejemplos es o no de la misma clase.

Procesamiento de imágenes

No todas las aplicaciones de las redes convolutivas están directamente relacionadas con visión artificial. Algunas de las aplicaciones más vistosas del *deep learning* son, en realidad, técnicas de procesamiento digital de imágenes:

- *Eliminación de fondos [background removal]*

Un problema de visión artificial relacionado con la detección de objetos es la segmentación semántica,⁷²⁷ en el que las redes convolutivas reciben como entrada una imagen y generan como salida otra imagen del mismo tamaño en la que aparecen etiquetados los píxeles de la imagen original. A diferencia de la segmentación a secas, que se limita a segmentar la imagen en partes coherentes, la segmentación semántica pretende comprender lo que representan esas partes. Por ejemplo, podemos utilizar técnicas de segmentación semántica para conseguir eliminar el fondo de una imagen de forma completamente automática (una versión sofisticada de la varita mágica que incluyen algunos programas de retoque fotográfico).

- *Manipulación de imágenes*

Los modelos generativos son modelos que pretenden sintetizar muestras que, sin aparecer en el conjunto de entrenamiento, tengan características similares a las de los ejemplos del conjunto de entrenamiento. Ya vimos que se pueden utilizar en modelos con adversario para mejorar la

⁷²⁷ Evan Shelhamer, Jonathan Long, y Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4): 640–651, 2017. ISSN 0162-8828. DOI: 10.1109/TPAMI.2016.2572683



Figura 154: Coloreado automático de imágenes: Un par de fotografías antiguas, en blanco y negro, junto con sus versiones coloreadas por una red neuronal artificial.

robustez de un modelo de aprendizaje automático. Aparte de ese uso, también se han empleado para generar imágenes fotorrealistas con resultados sorprendentes (p.ej. <http://www.evolvingai.org/ppgn>).⁷²⁸ Si el modelo generativo incluye parámetros a los que se pueda atribuir una interpretación semántica, esto abre la puerta a la posibilidad de manipular imágenes de formas hasta ahora imposibles. Por ejemplo, podríamos realizar operaciones “aritméticas” sobre imágenes del tipo ‘hombre con gafas’ - ‘hombre’ + ‘mujer’ = ‘mujer con gafas’, en las que hemos sintetizado la imagen de una mujer con gafas a partir de la imagen de un hombre con gafas.⁷²⁹ Otra manipulación posible consiste en modificar la edad de la persona que aparece en una fotografía, preservando la identidad de la persona pero modificando sus rasgos faciales.⁷³⁰ Usando este tipo de técnicas, existe el riesgo de que se facilite la creación de contenido multimedia falso: imágenes, sonidos o vídeos que un actor malintencionado puede utilizar para desinformar, ahora que están de moda las *fake news*.

⁷²⁸ Anh Nguyen, Jeff Clune, Yoshua Bengio, Alexey Dosovitskiy, y Jason Yosinski. Plug & play generative networks: Conditional iterative generation of images in latent space. En *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, páginas 3510–3520, 2017. ISBN 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.374

⁷²⁹ Alec Radford, Luke Metz, y Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *ICLR’2016*, arXiv:1511.06434, 2016. URL <http://arxiv.org/abs/1511.06434>

⁷³⁰ Grigory Antipov, Moez Baccouche, y Jean-Luc Dugelay. Face aging with conditional generative adversarial networks. *arXiv e-prints*, arXiv:1702.01983, 2017. URL <http://arxiv.org/abs/1702.01983>

- *Coloreado de imágenes [image colorization]*

Otra aplicación muy vistosa de las redes convolutivas es el coloreado de imágenes en blanco y negro. En vez de ir dibujando a mano sobre la imagen original, como se hacía para colorear películas antiguas, una red neuronal puede encargarse de realizar el trabajo por nosotros.⁷³¹ Entrenando una red convolutiva sobre millones de imágenes naturales, como en ImageNet, se puede conseguir que la red sea capaz de darle color a imágenes en escala de grises.

Eso sí, si en nuestro conjunto de datos predominan las personas de ojos marrones, es más que probable que la imagen coloreada le ponga los ojos marrones a quien los tenía azules. Para evitar problemas de este tipo, siempre se puede recurrir a una estrategia semiautomática en la que una herramienta interactiva permita al usuario indicarle al sistema el color que debería aparecer en determinadas zonas de la imagen, como iDeepColor (<https://richzhang.github.io/ideepcolor/>), lo que permite conseguir coloreados realistas sin demasiado esfuerzo.⁷³²

- *Super-resolución*

El objetivo de la super-resolución es, dada una imagen original de escasa resolución, producir una imagen de mayor resolución. El objetivo, claro está, es que la imagen de mayor resolución no aparezca pixelada, como si de un simple zoom se tratase, sino que sea lo más fotorrealista posible.

Se han propuesto diversas formas de lograrlo con redes neuronales convolutivas, ya sea definiendo una función de pérdida por píxel,⁷³³ usando técnicas sub-píxel,⁷³⁴ redes con adversario generativo [*GAN: Generative Adversarial Network*]⁷³⁵ o arquitecturas especializadas como PixelCNN.⁷³⁶ Igual que en otras tareas de procesamiento de imágenes, como el coloreado de imágenes en escala de grises, no hay una salida correcta, por lo que distintas estrategias pueden conducir a resultados visuales más que aceptables.

Las técnicas de super-resolución se pueden utilizar para mejorar automáticamente la calidad de imágenes captadas con sensores. Por ejemplo, se ha conseguido que las imágenes borrosas de galaxias lejanas captadas con un telescopio parezca que se tomaron con un telescopio mejor del que se usó. Una recomendación: en aplicaciones científicas, mejor andarse con mucho cuidado, no vaya a ser que la red le añada a la imagen detalles que realmente no estaban en el original.

- *Traducción de imagen a imagen [image-to-image translation]*

Algunas tareas de procesamiento de imágenes, como el coloreado de imágenes o la super-resolución, se pueden plantear como un problema de aprendizaje supervisado en el que el objetivo es aprender una función que establezca una correspondencia entre una imagen y otra.

Dado un conjunto de pares de imágenes como conjunto de entrena-

⁷³¹ Richard Zhang, Phillip Isola, y Alexei A. Efros. Colorful image colorization. *arXiv e-prints*, arXiv:1603.08511, 2016. URL <http://arxiv.org/abs/1603.08511>

⁷³² Richard Zhang, Jun-Yan Zhu, Phillip Isola, Xinyang Geng, Angela S. Lin, Tianhe Yu, y Alexei A. Efros. Real-time user-guided image colorization with learned deep priors. *ACM Transactions on Graphics*, 36(4):119:1–119:11, 2017b. DOI: 10.1145/3072959.3073703

⁷³³ Chao Dong, Chen Change Loy, Kai-ming He, , y Xiaoou Tang. Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2):295–307, 2016. ISSN 0162-8828. DOI: 10.1109/TPAMI.2015.2439281

⁷³⁴ Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, y Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. *arXiv e-prints*, arXiv:1609.05158, 2016. URL <http://arxiv.org/abs/1609.05158>

⁷³⁵ Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew P. Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, y Wenzhe Shi. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv e-prints*, arXiv:1609.04802, 2016. URL <http://arxiv.org/abs/1609.04802>

⁷³⁶ Ryan Dahl, Mohammad Norouzi, y Jonathon Shlens. Pixel recursive super resolution. *arXiv e-prints*, arXiv:1702.00783, 2017. URL <http://arxiv.org/abs/1702.00783>

miento, se puede entrenar una red convolutiva para que aprenda dicha correspondencia y luego sea capaz de aplicarla a nuevas imágenes. Éste es el planteamiento de la traducción de imagen a imagen, un enfoque genérico que sirve para sintetizar fotografías a partir de mapas etiquetados, reconstruir objetos a partir de sus contornos geométricos o colorear imágenes. Los resultados son, de nuevo, espectaculares, y el software necesario está disponible en Internet, p.ej. pix2pix.⁷³⁷

Para muchas tareas de interés, puede que no dispongamos de un conjunto de entrenamiento adecuado. Incluso en esos casos se puede recurrir a las técnicas de traducción de imagen a imagen. Una vez más, los modelos con adversario pueden rescatarnos.⁷³⁸ Con técnicas así, además de mejorar la calidad de nuestras fotografías, se puede, por ejemplo, hacer que una fotografía tomada en verano parezca que la tomamos en invierno (o viceversa).

Síntesis de imágenes: Arte neuronal

Para Mark Riedl, existe una forma más rigurosa de evaluar la inteligencia potencial de una máquina que el conocido test de Turing: el test de inteligencia y creatividad artificial de Lovelace. El test original de Lovelace, propuesto en 2001, consistía en hacer que el agente artificial crease algo valioso, novedoso y sorprendente cuya producción su diseñador no pudiese explicar. Su versión 2.0 intenta ser menos subjetiva, especificando los criterios que debe seguir el evaluador humano para evaluar el trabajo creativo de la máquina sin realizar valoraciones de juicio.⁷³⁹

Aunque no superen el test de creatividad de Lovelace, existen aplicaciones del *deep learning* que sí pueden resultar sorprendentes. La separación de forma y contenido en imágenes es una de esas aplicaciones de las redes convolutivas que podríamos llamar poco convencional.⁷⁴⁰

La idea es muy elegante y, hasta cierto punto, similar a la del uso de redes siamesas. Imaginemos que partimos de una red convolutiva ya entrenada sobre un conjunto de datos enorme (como las redes VGG de la competición de ImageNet). Ahora, nos olvidamos de la parte de la red que se encarga de resolver un problema de clasificación de imágenes o de detección de objetos. Nos quedamos únicamente con sus capas convolutivas, las que extraen características de la imagen de entrada. A continuación, utilizamos esas capas de la red con dos imágenes diferentes:

- *Contenido:* La primera imagen, c , contiene una escena visual cuyo estilo queremos modificar (p.ej. una fotografía de nuestras últimas vacaciones).
- *Estilo:* La segunda imagen, s , tiene un estilo visual que nos resulta atractivo (p.ej. algún cuadro de un pintor, famoso o no tan famoso).

⁷³⁷ Phillip Isola, Jun-Yan Zhu, Ting-hui Zhou, y Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *arXiv e-prints*, arXiv:1611.07004, 2016. URL <http://arxiv.org/abs/1611.07004>

⁷³⁸ Jun-Yan Zhu, Taesung Park, Phillip Isola, y Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv e-prints*, arXiv:1703.10593, 2017. URL <http://arxiv.org/abs/1703.10593>

⁷³⁹ Mark O. Riedl. The lovelace 2.0 test of artificial creativity and intelligence. *arXiv e-prints*, arXiv:1410.6142, 2014. URL <http://arxiv.org/abs/1410.6142>

⁷⁴⁰ Alejandro Avilés. Form and Content in Neural Networks: Unconventional Applications of Deep Learning. Master's thesis, Department of Computer Science, University of Granada, Spain, 2016

Cuando, a la red convolutiva, le proporcionamos ambas imágenes de entrada, cada una de ellas generará una serie de patrones de actividad en las neuronas de las capas ocultas. Para darle a la primera imagen el estilo de la segunda, generaremos una nueva imagen x en la que intentamos, simultáneamente reproducir tanto el contenido de c como el estilo de s . Para ello, definimos dos funciones de coste o pérdida:⁷⁴¹

- Una función de pérdida, $\mathcal{L}_{contenido}$, que intente reproducir, en la imagen sintetizada, x , el contenido de la imagen original c , intentando lograr que la imagen sintética genere las mismas activaciones, f_x , que las activaciones de la imagen de partida, f_c :

$$\mathcal{L}_{contenido}(x, c) = \sum (f_x^2 - f_c^2)$$

- Una función de pérdida, \mathcal{L}_{estilo} , que intente reproducir, en la imagen sintetizada, x , el estilo de la imagen s . Esta función es algo más difícil de concretar. Inspirados por el trabajo de Javier Portilla y Eero Simoncelli,⁷⁴² decidimos que el estilo puede representarse por la correlación existente entre los niveles de activación de las diferentes neuronas que se encuentran en la misma capa de la red.

Para obtener esa correlación, calculamos la matriz grammiana G a partir de los niveles de activación de las neuronas de la misma capa (repitiendo el proceso para las diferentes capas de la red). El elemento (i, j) de la matriz es, simplemente, el producto vectorial de los niveles de activación de las neuronas correspondientes al píxel (i, j) , teniendo en cuenta que las capas convolutivas tienen profundidad (varios canales):

$$G(i, j) = \sum_k f(i, k) f(j, k)$$

Ese cálculo lo hacemos, tanto para la imagen sintetizada, G_x , como para la imagen de la que queremos tomar prestado el estilo, G_s . Para cada capa l de la red, su contribución a la función de pérdida será

$$\mathcal{L}_{estilo}(x, s, l) = \frac{1}{4N^2M^2} \sum_{i,j} (G_x(i, j) - G_s(i, j))^2$$

Para obtener la función de pérdida asociada al estilo, combinamos las funciones de pérdida asociadas a cada capa:

$$\mathcal{L}_{estilo}(x, s) = \sum_l w_l \mathcal{L}_{estilo}(x, s, l)$$

donde los parámetros w_l los podemos utilizar para ajustar la contribución de cada capa a la función de pérdida total (el mismo peso para las distintas capas en una red como VGG suele funcionar correctamente).

Una vez que tenemos una representación cuantitativa del parecido de la imagen sintetizada con el contenido de la imagen original, $\mathcal{L}_{contenido}(x, c)$,

⁷⁴¹ Leon A. Gatys, Alexander S. Ecker, y Matthias Bethge. Image style transfer using convolutional neural networks. En *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2414–2423, 2016b. ISBN 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.265

⁷⁴² Javier Portilla y Eero P. Simoncelli. A parametric texture model based on joint statistics of complex wavelet coefficients. *International Journal of Computer Vision*, 40(1):49–70, 2000. ISSN 1573-1405. DOI: 10.1023/A:1026553619983

y del parecido entre los estilos de la imagen sintetizada y de la imagen de la que queremos tomar prestado el estilo, $\mathcal{L}_{estilo}(x, s)$, las podemos combinar en una función de pérdida total:

$$\mathcal{L}_{total}(x, c, s) = \alpha \mathcal{L}_{contenido}(x, c) + \beta \mathcal{L}_{estilo}(x, s)$$

donde los parámetros α y β nos permiten ponderar la reconstrucción de contenido y estilo, respectivamente.

Para realizar la transferencia de estilo de una imagen a otra sólo tenemos que darle un giro a la forma en la que tradicionalmente trabajamos con redes neuronales. Cuando entrenamos una red, asumimos que nos proporcionan una entrada, que es fija, y nuestra misión es ajustar sus parámetros para reducir el error observado en la salida. Cuando transferimos estilos, asumimos que los pesos de la red son fijos (los de una red convolutiva ya entrenada) y ajustamos su entrada para minimizar la función de pérdida anterior. Sólo tenemos que partir de ruido aleatorio en la entrada y dejar que el mismo proceso iterativo de optimización que normalmente utilizamos para ajustar los parámetros de la red, sea capaz de ajustar la entrada de la red hasta conseguir una imagen sintetizada en la que su contenido y su estilo provienen de las dos imágenes de partida.

En ocasiones, al transformar el estilo de una imagen tomando los colores de otra, podemos estar modificando la apariencia de la imagen original de una forma no deseable. De forma sencilla, podemos prevenir cambios no deseados en el color de la imagen original,⁷⁴³ así como controlar algunas cualidades perceptivas de la imagen para que el resultado final resulte estéticamente agradable⁷⁴⁴ o consiga ser fotorrealista.⁷⁴⁵

Aunque el proceso descrito de optimización es computacionalmente costoso, se puede agilizar si fijamos el estilo y creamos una red que aplique ese estilo a cualquier imagen que nos llegue.⁷⁴⁶ Esto nos permite realizar transferencias de estilo en tiempo real, algo que podemos hacer, con nuestras propias imágenes, en algunas páginas web (<https://deepoch.io/>), usando bots de Twitter (@DeepForger) o con apps para móviles (<https://prisma-ai.com/>).

Esta misma técnica se ha empleado para aplicar estilos en algunas escenas en cortos de cine (*Come Swim*, Kristen Stewart, 2017),⁷⁴⁷ sin necesidad de contratar a cientos de dibujantes que vayan aplicando un estilo fotograma a fotograma (como en *Loving Vincent*, Dorota Kobiela y Hugh Welchman, 2017). Facebook también ha experimentado con esta técnica para aplicársela, en tiempo real, a vídeos de sus usuarios.

Un proceso similar de transferencia de estilos se puede emplear para otros tipos de señales, como señales de audio. Usando otro tipo de redes neuronales, DeepBach es capaz de modelar la música polifónica de Johann Sebastian Bach y generar melodías con el estilo del compositor de las variaciones Goldberg y los conciertos de Brandenburg.⁷⁴⁸

⁷⁴³ Leon A. Gatys, Matthias Bethge, Aaron Hertzmann, y Eli Shechtman. Preserving color in neural artistic style transfer. *arXiv e-prints*, arXiv:1606.05897, 2016a. URL <http://arxiv.org/abs/1606.05897>

⁷⁴⁴ Leon A. Gatys, Alexander S. Ecker, Matthias Bethge, Aaron Hertzmann, y Eli Shechtman. Controlling perceptual factors in neural style transfer. En *CVPR'2017*, pages 3730–3738, 2017. ISBN 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.397

⁷⁴⁵ Fujun Luan, Sylvain Paris, Eli Shechtman, y Kavita Bala. Deep photo style transfer. *arXiv e-prints*, arXiv:1703.07511, 2017. URL <http://arxiv.org/abs/1703.07511>

⁷⁴⁶ Justin Johnson, Alexandre Alahi, y Fei-Fei Li. Perceptual losses for real-time style transfer and super-resolution. *arXiv e-prints*, arXiv:1603.08155, 2016a. URL <http://arxiv.org/abs/1603.08155>

⁷⁴⁷ Bhautik J. Joshi, Kristen Stewart, y David Shapiro. Bringing impressionism to life with neural style transfer in come swim. *arXiv e-prints*, arXiv:1701.04928, 2017. URL <http://arxiv.org/abs/1701.04928>

⁷⁴⁸ Gaëtan Hadjeres y François Fleuret. DeepBach: a Steerable Model for Bach chorales generation. *arXiv*, arXiv:1612.01010, 2016. URL <http://arxiv.org/abs/1612.01010>

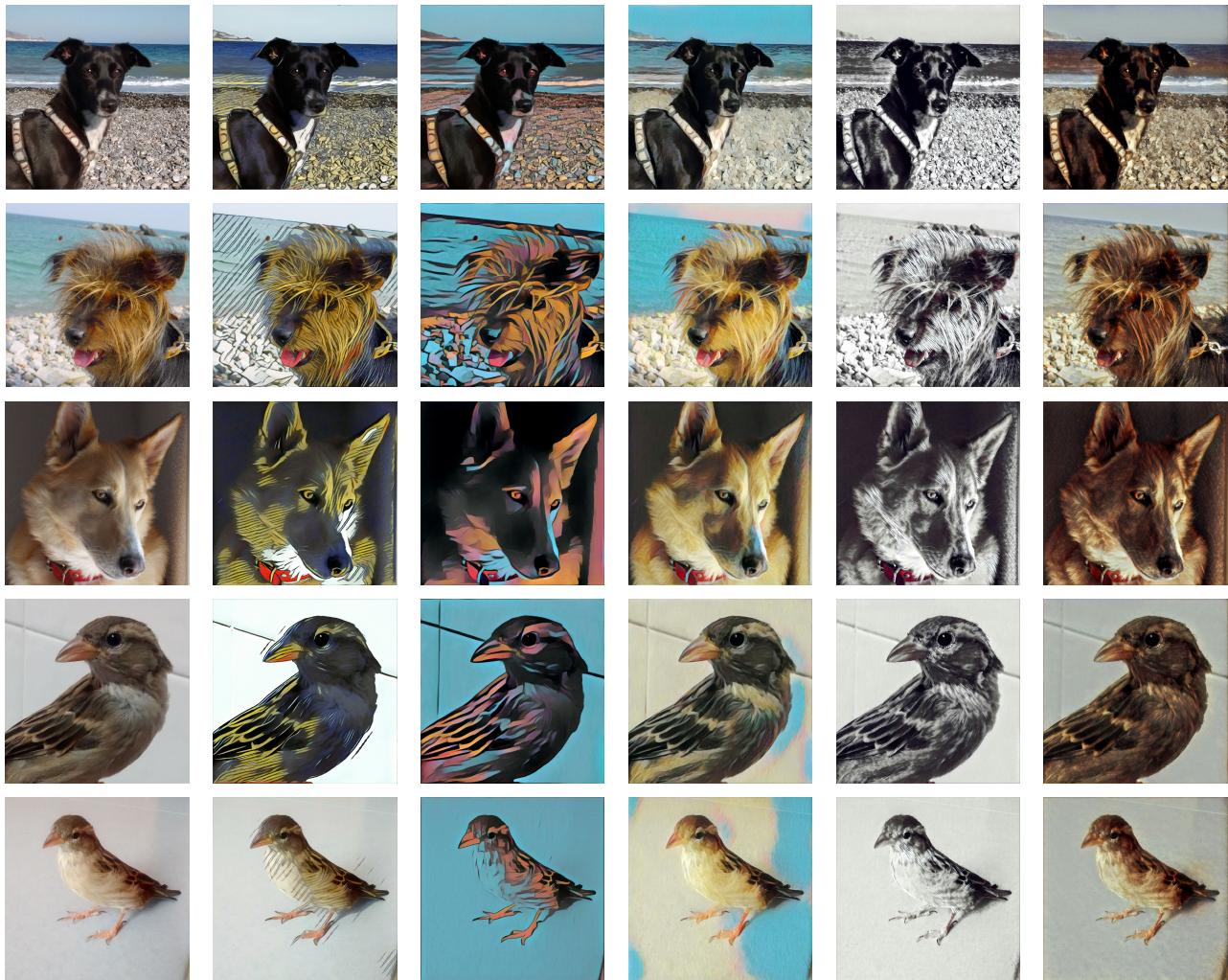


Figura 155: Transferencia de estilos:
Imágenes originales y cinco estilos.



Figura 156: Transferencia de estilos: Otros seis estilos más (incluyendo los correspondientes a algún que otro cuadro famoso).



Figura 157: Transferencia de estilos:
Múltiples estilos aplicados sobre una
misma imagen.

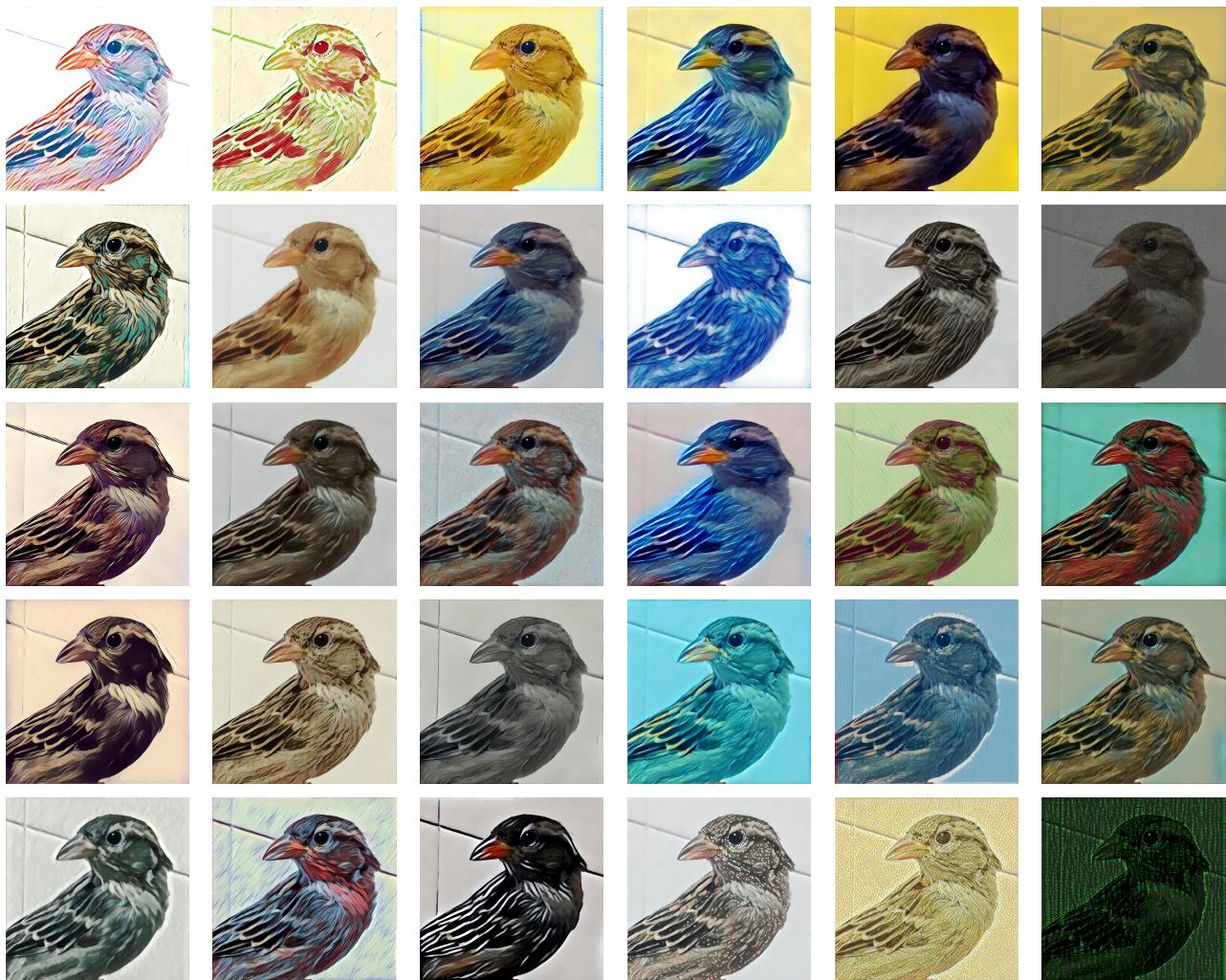


Figura 158: Transferencia de estilos:
Múltiples estilos aplicados sobre una
misma imagen (continuación).

No son éstas las únicas aplicaciones artísticas de las redes neuronales artificiales:

- Intentando analizar el funcionamiento de la red Inception usada en la competición ImageNet, los ingenieros de Google se dieron cuenta de que las redes entrenadas para discriminar entre varias categorías de imágenes contienen también gran cantidad de información que se puede utilizar para generar nuevas imágenes. Esta observación dio lugar a un nuevo estilo artístico, cómicamente denominado “inceptionismo”.⁷⁴⁹ El inceptionismo está caracterizado por el uso de imágenes deliberadamente procesadas más de la cuenta, lo que da lugar a escenas alucinógenas que parecen sacadas de un sueño. De ahí el nombre de DeepDream, el sistema de Google que acerca el *deep learning* al mundo de los sueños. Al usar DeepDream, en lugar de detectar patrones en una imagen, se genera una imagen a partir de los patrones detectados en la imagen original. Repitiendo el proceso iterativamente, se obtienen resultados sobrecojedores...
- Sin tantas ensoñaciones, investigadores de Adobe y de la Universidad de California en Berkeley trabajan en desarrollar software que automáticamente genere imágenes detalladas inspiradas por el color y la forma de los “brochazos” del usuario.⁷⁵⁰ El desarrollo de software interactivo de este tipo abre nuevas posibilidades para el proceso de creación de obras de arte.

⁷⁴⁹ Alexander Mordvintsev, Christopher Olah, y Mike Tyka. Inceptionism: Going Deeper into Neural Networks. Technical report, Google Research Blog, 2015. URL <https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html>

⁷⁵⁰ Jun-Yan Zhu, Philipp Krähenbühl, Eli Shechtman, y Alexei A. Efros. Generative visual manipulation on the natural image manifold. *arXiv e-prints*, arXiv:1609.03552, 2016. URL <http://arxiv.org/abs/1609.03552>

Bibliografía

Scott Aaronson. *Quantum Computing since Democritus*. Cambridge University Press, 2013. ISBN 0521199565.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, y Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv e-prints*, arXiv:1603.04467, 2016. URL <http://arxiv.org/abs/1603.04467>.

Ossama Abdel-Hamid, Li Deng, y Dong Yu. Exploring Convolutional Neural Network Structures and Optimization Techniques for Speech Recognition. En *INTERSPEECH 2013, 14th Annual Conference of the International Speech Communication Association, Lyon, France, August 25-29, 2013*, pages 3366–3370. ISCA, 2013. URL <https://doi.org/10.1109/TASLP.2014.2339736>.

Ossama Abdel-Hamid, Abdel rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, y Dong Yu. Convolutional Neural Networks for Speech Recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(10):1533–1545, 2014. ISSN 2329-9290. DOI: 10.1109/TASLP.2014.2339736.

S. Abid, F. Fnaiech, y M. Najim. A fast feedforward training algorithm using a modified form of the standard backpropagation algorithm. *IEEE Transactions on Neural Networks*, 12(2):424–430, Mar 2001. ISSN 1045-9227. DOI: 10.1109/72.914537.

Yaser S. Abu-Mostafa. Learning from hints. *Journal of Complexity*, 10(1):165 – 178, 1994. ISSN 0885-064X. DOI: jcom.1994.1007.

Yaser S. Abu-Mostafa. Hints. *Neural Computation*, 7(4):639–671, 1995.
 DOI: 10.1162/neco.1995.7.4.639.

Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Pan Books, 1979. ISBN 0330258648.

Ernest Adams y Joris Dormans. *Game Mechanics: Advanced Game Design*. New Riders Publishing, 2012. ISBN 0321820274.

Charu C. Aggarwal. *Outlier Analysis*. Springer, 2013. ISBN 1461463955.

Charu C. Aggarwal. *Recommender Systems: The Textbook*. Springer, 1st edition, 2016. ISBN 3319296574.

Filipe Aires, Michel Schmitt, Alain Chedin, , y Noelle Scott. The weight smoothing regularization of MLP for Jacobian stabilization. *IEEE Transactions on Neural Networks*, 10(6):1502–1510, Nov 1999. ISSN 1045-9227. DOI: 10.1109/72.809096.

Igor N. Aizenberg, Naum N. Aizenberg, y Joos P. Vandewalle. *Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications*. Kluwer Academic Publishers, 2000. ISBN 0792378245.

Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülcöhre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Si-na Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban,

Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, y Ying Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, arXiv:1605.02688, 2016. URL <http://arxiv.org/abs/1605.02688>.

Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, 3rd edition, 2014. ISBN 0262028182.

Ganesh Ananthanarayanan y Ishai Menache. *Big Data Analytics Systems*, pages 137–160. Cambridge University Press, 2016. ISBN 1107099005.

Aristoklis D. Anastasiadis, George D. Magoulas, y Michael N. Vrahatis. New globally convergent training scheme based on the resilient propagation algorithm. *Neurocomputing*, 64:253 – 270, 2005. ISSN 0925-2312. DOI: 10.1016/j.neucom.2004.11.016. Trends in Neurocomputing: 12th European Symposium on Artificial Neural Networks 2004.

James A. Anderson y Edward Rosenfeld, editores. *Neurocomputing: Foundations of Research*. MIT Press, 1988. ISBN 0262010976.

James A. Anderson, Andreas Pellionisz, y Edward Rosenfeld, editores. *Neurocomputing: Directions for Research*. MIT Press, 1990. ISBN 0262011190.

Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, y Nando de Freitas. Learning to learn by gradient descent by gradient descent. *arXiv e-prints*, arXiv:1606.04474, 2016. URL <http://arxiv.org/abs/1606.04474>.

J. K. Anlauf y M. Biehl. Erratum: The AdaTron: An Adaptive Perceptron Algorithm. *EPL (Europhysics Letters)*, 11(4):387, 1990. URL <http://stacks.iop.org/0295-5075/11/i=4/a=016>.

J. K. Anlauf y Michael Biehl. The AdaTron: An Adaptive Perceptron Algorithm. *EPL (Europhysics Letters)*, 10(7):687, 1989. URL <http://stacks.iop.org/0295-5075/10/i=7/a=014>.

Grigory Antipov, Moez Baccouche, y Jean-Luc Dugelay. Face aging with conditional generative adversarial networks. *arXiv e-prints*, arXiv:1702.01983, 2017. URL <http://arxiv.org/abs/1702.01983>.

Michael A. Arbib y James J. Bonaiuto, editores. *From Neuron to Cognition via Computational Neuroscience*. MIT Press, 2016. ISBN 0262034964.

Dan Ariely. *Predictably Irrational: The Hidden Forces That Shape Our Decisions*. Harper, 2009. ISBN 0061854549.

Larry Armijo. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics*, 16(1):1–3, 1966. ISSN 0030-8730. URL <https://msp.org/pjm/1966/16-1/p01.xhtml>.

Stefan Arnborg, Derek G. Corneil, y Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987. ISSN 0196-5212. DOI: 10.1137/0608024.

William Ross Ashby. *Design for a Brain*. Chapman and Hall, 1952. URL <https://archive.org/details/designforbrain00ashb>.

Alejandro Avilés. Form and Content in Neural Networks: Unconventional Applications of Deep Learning. Master’s thesis, Department of Computer Science, University of Granada, Spain, 2016.

Francis R. Bach y David M. Blei, editores. *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, 2015. JMLR.org. URL <http://jmlr.org/proceedings/papers/v37/>.

Dzmitry Bahdanau, Kyunghyun Cho, y Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *ICLR’2015*, arXiv:1409.0473, 2015. URL <http://arxiv.org/abs/1409.0473>.

Pierre Baldi y Peter J Sadowski. Understanding dropout. En C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, y K. Q. Weinberger, editores, *NIPS’2013 Advances in Neural Information Processing Systems 26*, pages 2814–2822. Curran Associates, Inc., 2013. URL <https://goo.gl/g4ocCU>.

Dana H. Ballard. *Brain Computation as Hierarchical Abstraction*. MIT Press, 2015. ISBN 0262028611.

Michele Banko y Eric Brill. Scaling to very very large corpora for natural language disambiguation. En *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL ’01, pages 26–33, Stroudsburg, PA, USA, 2001. Association for Computational Linguistics. DOI: 10.3115/1073012.1073017.

Andrew R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, May 1993. ISSN 0018-9448. DOI: 10.1109/18.256500.

Roberto Battiti. Accelerated Backpropagation Learning: Two Optimization Methods. *Complex Systems*, 3(4):331–342, 1980. ISSN 0891-2513. URL http://www.complex-systems.com/abstracts/v03_i04_a02.html.

Roberto Battiti. First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method. *Neural Computation*, 4(2):141–166, 1992. DOI: 10.1162/neco.1992.4.2.141.

Roberto Battiti y Francesco Masulli. BFGS Optimization for Faster and Automated Supervised Learning. En *International Neural Network Conference: July 9–13, 1990 Palais Des Congres, Paris, France*, pages 757–760, 1990. ISBN 978-94-009-0643-3. DOI: 10.1007/978-94-009-0643-3_68.

Roberto Battiti y Giampietro Tecchiolli. Learning with first, second, and no derivatives: A case study in high energy physics. *Neurocomputing*, 6(2):181 – 206, 1994. ISSN 0925-2312. DOI: 10.1016/0925-2312(94)90054-X.

Eric B. Baum y Frank Wilczek. Supervised learning of probability distributions by neural networks. En D. Z. Anderson, editor, *NIPS'1987 Neural Information Processing Systems*, pages 52–61. American Institute of Physics, 1988. URL <https://goo.gl/v9TrQU>.

Jonathan Baxter. Learning internal representations. En *Proceedings of the Eighth Annual Conference on Computational Learning Theory*, COLT '95, pages 311–320, 1995. ISBN 0897917235. DOI: 10.1145/225298.225336.

Russell Beale y Tom Jackson. *Neural Computing - An Introduction*. IOP Publishing, Taylor & Francis Group, 1990. ISBN 0852742622.

Randall D. Beer y John C. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91–122, 1992. DOI: 10.1177/105971239200100105.

Homayoon S.M. Beigi. Neural network learning through optimally conditioned quadratically convergent methods requiring no line search. En *Proceedings of 36th Midwest Symposium on Circuits and Systems*, pages 109–112 vol.1, Aug 1993. DOI: 10.1109/MWSCAS.1993.343053.

Samy Bengio, Yoshua Bengio, y Jocelyn Cloutier. On the search for new learning rules for ANNs. *Neural Processing Letters*, 2(4):26–30, 1995. ISSN 1573-773X. DOI: 10.1007/BF02279935.

Yoshua Bengio. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, January 2009. ISSN 1935-8237. DOI: 10.1561/2200000006.

Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. En Grégoire Montavon, Geneviève B. Orr, y Klaus-Robert Müller, editores, *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2nd edition, 2012a. ISBN 364235288X. DOI: 10.1007/978-3-642-35289-8_26.

Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *arXiv e-prints*, arXiv:1206.5533, 2012b. URL <http://arxiv.org/abs/1206.5533>.

Yoshua Bengio. Deep Learning: Theoretical Motivations. *Deep Learning Summer School, Montreal*, 2015. URL <https://goo.gl/pFXok6>.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, y Christian Janvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, March 2003. ISSN 1532-4435. URL <http://www.jmlr.org/papers/v3/bengio03a.html>.

Yoshua Bengio, Pascal Lamblin, Dan Popovici, y Hugo Larochelle. Greedy layer-wise training of deep networks. En *NIPS’2006 Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 153–160, 2006a. URL <https://goo.gl/KyBc9x>.

Yoshua Bengio, Nicolas L. Roux, Pascal Vincent, Olivier Delalleau, y Patrice Marcotte. Convex neural networks. En Y. Weiss, P. B. Schölkopf, y J. C. Platt, editores, *NIPS’2005 Advances in Neural Information Processing Systems 18*, pages 123–130. MIT Press, 2006b. URL <https://goo.gl/DsBURU>.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, y Jason Weston. Curriculum learning. En Andrea Pohoreckyj Danyluk, Léon Bottou, y Michael L. Littman, editores, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 41–48. ACM, 2009. ISBN 978-1-60558-516-1. DOI: 10.1145/1553374.1553380.

Yoshua Bengio, Aaron C. Courville, y Pascal Vincent. Unsupervised Feature Learning and Deep Learning: A Review and New Perspectives. *arXiv e-prints*, arXiv:1206.5538, 2012. URL <http://arxiv.org/abs/1206.5538>.

Yoshua Bengio, Nicolas Boulanger-Lewandowski, y Razvan Pascanu. Advances in optimizing recurrent networks. En *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8624–8628, May 2013a. DOI: 10.1109/ICASSP.2013.6639349.

Yoshua Bengio, Aaron Courville, y Pascal Vincent. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, Aug 2013b. ISSN 0162-8828. DOI: 10.1109/TPAMI.2013.50.

James Bergstra y Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13(1):281–305, February 2012. ISSN 1532-4435. URL <http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>.

James S. Bergstra, Rémi Bardenet, Yoshua Bengio, y Balázs Kégl. Algorithms for hyper-parameter optimization. En J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *NIPS’2011 Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011. URL <https://goo.gl/mBi6jg>.

Hans J. Berliner. Backgammon Computer Program Beats World Champion. *Artificial Intelligence*, 14(2):205–220, 1980. DOI: 10.1016/0004-3702(80)90041-7.

Gregory Berns. *How Dogs Love Us: A Neuroscientist and His Adopted Dog Decode the Canine Brain*. New Harvest, 2013. ISBN 0544114515.

Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific Publishers, 2nd edition, 1999. ISBN 1886529000.

Fernando Berzal, Juan Carlos Cubero, Fernando Cuenca, y María José Martín-Bautista. On the quest for easy-to-understand splitting rules. *Data and Knowledge Engineering*, 44(1):31–48, 2003. DOI: 10.1016/S0169-023X(02)00062-9.

Fernando Berzal, Juan Carlos Cubero, Daniel Sánchez, y José María Serrano. ART: A Hybrid Classification Model. *Machine Learning*, 54(1):67–92, 2004. ISSN 1573-0565. DOI: 10.1023/B:MACH.0000008085.22487.a6.

Amit Bhaya y Eugenius Kaszkurewicz. Steepest descent with momentum for quadratic functions is a version of the conjugate gradient method. *Neural Networks*, 17(1):65 – 71, 2004. ISSN 0893-6080. DOI: 10.1016/S0893-6080(03)00170-9.

Monica Bianchini y Franco Scarselli. On the complexity of neural network classifiers: A comparison between shallow and deep architectures. *IEEE Transactions on Neural Networks and Learning Systems*, 25(8):1553–1565, 2014. DOI: 10.1109/TNNLS.2013.2293637.

Elie L. Bienenstock, Leon N. Cooper, y Paul W. Munro. Theory for the development of neuron selectivity: orientation specificity and binocular

interaction in visual cortex. *The Journal of Neuroscience*, 2(1):32–48, 1982. ISSN 0270-6474. URL <http://jneurosci.org/content/2/1/32>.

Christopher M. Bishop. Exact Calculation of the Hessian Matrix for the Multilayer Perceptron. *Neural Computation*, 4(4):494–501, 1992. DOI: 10.1162/neco.1992.4.4.494.

Christopher M. Bishop. Training with Noise is Equivalent to Tikhonov Regularization. *Neural Computation*, 7(1):108–116, January 1995a. ISSN 0899-7667. DOI: 10.1162/neco.1995.7.1.108.

Christopher M. Bishop. Regularization and complexity control in feed-forward networks. En F. Fougeisan-Soulie y P. Gallinari, editores, *Proceedings International Conference on Artificial Neural Networks ICANN'95*, volume 1, pages 141–148. EC2 et Cie, January 1995b. URL <https://goo.gl/ZKBTvj>.

Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1996. ISBN 0198538642.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, 2006. ISBN 0387310738.

David M. Blei, Andrew Y. Ng, y Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, March 2003. ISSN 1532-4435. URL <http://jmlr.csail.mit.edu/papers/v3/blei03a.html>.

Timothy Vivian Pelham Bliss y Terje Lomo. Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *The Journal of Physiology*, 232(2):331–356, 1973. ISSN 1469-7793. DOI: 10.1113/jphysiol.1973.sp010273.

Henry David Block. The Perceptron: A Model for Brain Functioning. I. *Reviews of Modern Physics*, 34:123–135, Jan 1962. DOI: 10.1103/RevModPhys.34.123.

Avrim Blum y Ronald L. Rivest. Training a 3-Node Neural Network is NP-Complete. En D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 494–501. Morgan-Kaufmann, 1989. URL <https://goo.gl/J1dVHj>.

Avrim Blum y Ronald L. Rivest. Training a 3-node neural network is NP-complete. *Neural Networks*, 5(1):117–127, 1992. DOI: 10.1016/S0893-6080(05)80010-3.

Julius R. Blum. Multidimensional stochastic approximation methods. *The Annals of Mathematical Statistics*, 25(4):737–744, 12 1954. DOI: 10.1214/aoms/1177728659.

Egbert J. W. Boers y Ida G. Sprinkhuizen-Kuyper. Biological metaphors and the design of modular artificial neural networks. Master's thesis, Department of Computer Science, Leiden University, The Netherlands, 1992.

Egbert J. W. Boers y Ida G. Sprinkhuizen-Kuyper. Combined Biological Metaphors. En Mukesh Patel, Vasant Honavar, y Karthik Balakrishnan, editores, *Advances in the Evolutionary Synthesis of Intelligent Agents*, pages 153—183. MIT Press, 2001. ISBN 0262162016.

Antoine Bordes, Léon Bottou, y Patrick Gallinari. SGD-QN: Careful Quasi-Newton Stochastic Gradient Descent. *J. Mach. Learn. Res.*, 10: 1737–1754, December 2009. ISSN 1532-4435. URL <http://www.jmlr.org/papers/volume10/bordes09a/bordes09a.pdf>.

Nick Bostrom. *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press, 1st edition, 2014. ISBN 0199678111.

Léon Bottou. Online Algorithms and Stochastic Approximations. En David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. ISBN 0521652634. URL <http://leon.bottou.org/publications/pdf/online-1998.pdf>. Revisado en junio de 2017.

Léon Bottou. Multilayer Neural Networks. *Deep Learning Summer School, Montreal*, 2012. URL http://videolectures.net/deeplearning2015_bottou_neural_networks/.

Léon Bottou y Olivier Bousquet. The tradeoffs of large scale learning. En *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pages 161–168, USA, 2007. Curran Associates Inc. ISBN 978-1-60560-352-0. URL <https://goo.gl/CSz6jZ>.

Léon Bottou, Frank E. Curtis, y Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning. *arXiv e-prints*, arXiv:1606.04838v2, 2017. URL <http://arxiv.org/abs/1606.04838>.

Y-Lan Boureau, Jean Ponce, y Yann LeCun. A theoretical analysis of feature pooling in visual recognition. En *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 111–118, USA, 2010. Omnipress. ISBN 978-1-60558-907-7.

Y-Lan Boureau, Nicolas Le Roux, Francis Bach, Jean Ponce, y Yann LeCun. Ask the locals: Multi-way local pooling for image recognition. En *2011 International Conference on Computer Vision*, pages 2651–2658, 2011. doi: 10.1109/ICCV.2011.6126555.

George E. P. Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976. DOI: 10.1080/01621459.1976.10480949.

George E. P. Box. Robustness in the strategy of scientific model building. En Robert L. Launer y Graham N Wilkinson, editores, *Robustness in Statistics*, pages 201–236. Academic Press, 1979. ISBN 0124381502.

George E. P. Box y Mervin E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 06 1958. DOI: 10.1214/aoms/1177706645.

Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, August 1996. ISSN 0885-6125. DOI: 10.1023/A:1018054314350.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, October 2001. ISSN 0885-6125. DOI: 10.1023/A:1010933404324.

Richard P. Brent. *Algorithms for Minimization without Derivatives*. Prentice-Hall, 1973. ISBN 0130223352.

John S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. En D. S. Touretzky, editor, *NIPS'1989 Advances in Neural Information Processing Systems 2*, pages 211–217. Morgan-Kaufmann, 1990a. URL <https://goo.gl/Y65Rdr>.

John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. En Françoise Fogelman Soulié y Jeanny Hérault, editores, *Neurocomputing: Algorithms, Architectures and Applications*, volume 98 of *NATO ASI Series*, pages 227–236. Springer, Berlin, Heidelberg, 1990b. ISBN 978-3-642-76153-9. DOI: 10.1007/978-3-642-76153-9_28.

Charles George Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation*, 19(92):577–593, 1965. ISSN 0025-5718. DOI: 10.1090/S0025-5718-1965-0198670-6. URL <http://www.jstor.org/stable/2003941>.

Charles George Broyden. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970. ISSN 0272-4960. DOI: 10.1093/imamat/6.1.76.

Arthur Earl Bryson. A gradient method for optimizing multi-stage allocation processes. En *Proceedings of the Harvard University Symposium on Digital Computers and Their Applications*, April 1961.

Arthur Earl Bryson y Yu-Chi Ho. *Applied Optimal Control: Optimization, Estimation, and Control*. Blaisdell Publishing Company or Xerox College Publishing, 1969.

Arthur Earl Bryson y Yu-Chi Ho. *Applied Optimal Control: Optimization, Estimation, and Control*. John Wiley & Sons, revised edition, 1975. ISBN 0470114819.

Arthur Earl Bryson, W.F. Denham, y Stuart E. Dreyfus. Optimal programming problems with inequality constraints. i: Necessary conditions for extremal solutions. *AIAA Journal*, 1(11):2544–2550, 1963. DOI: 10.2514/3.2107.

Cristian Bucilă, Rich Caruana, y Alexandru Niculescu-Mizil. Model compression. En *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 535–541, 2006. ISBN 1-59593-339-5. DOI: 10.1145/1150402.1150464.

Wray Buntine. Inductive knowledge acquisition and induction methodologies. *Knowledge-Based Systems*, 2(1):52–61, March 1989. ISSN 0950-7051. DOI: 10.1016/0950-7051(89)90008-7.

Richard H. Byrd, Peihuang Lu, Jorge Nocedal, y Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995. DOI: 10.1137/0916069.

Robert Callan. *The Essence of Neural Networks*. Prentice Hall (UK), 1999. ISBN 013908732X.

Murray Campbell, A. Joseph Hoane, y Feng-Hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1):57 – 83, 2002. ISSN 0004-3702. DOI: 10.1016/S0004-3702(01)00129-1. URL <http://www.sciencedirect.com/science/article/pii/S0004370201001291>.

Alfredo Canziani, Adam Paszke, y Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv e-prints*, arXiv:1605.07678, 2016. URL <http://arxiv.org/abs/1605.07678>.

Gail A. Carpenter y Stephen Grossberg. The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network. *Computer*, 21(3): 77–88, 1988. ISSN 0018-9162. DOI: 10.1109/2.33.

Richard A. Caruana. Multitask connectionist learning. En *In Proceedings of the 1993 Connectionist Models Summer School*, pages 372–379, 1993.

Richard A. Caruana. Multitask learning. *Machine Learning*, 28(1): 41–75, Jul 1997. ISSN 1573-0565. DOI: 10.1023/A:1007379606734.

Augustin-Louis Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Compte Rendu des Séances de L'Académie des Sciences XXV*, Série A(25):536–538, 1847.

Maureen Caudill y Charles Butler. *Naturally Intelligent Systems*. MIT Press, 1st edition, 1990. ISBN 0262031566.

Pravin Chandra y Yogesh Singh. Regularization and feedforward artificial neural network training with noise. En *Proceedings of the 2003 International Joint Conference on Neural Networks*, volume 3, pages 2366–2371 vol.3, July 2003. DOI: 10.1109/IJCNN.2003.1223782.

Pravin Chandra y Yogesh Singh. An activation function adapting training algorithm for sigmoidal feedforward networks. *Neurocomputing*, 61: 429 – 437, 2004. ISSN 0925-2312. DOI: 10.1016/j.neucom.2004.04.001. Hybrid Neurocomputing: Selected Papers from the 2nd International Conference on Hybrid Intelligent Systems.

Hung-Han Chen, Michael T. Manry, y Hema Chandrasekaran. A neural network training algorithm utilizing multiple sets of linear equations. *Neurocomputing*, 25(1):55 – 72, 1999. ISSN 0925-2312. DOI: 10.1016/S0925-2312(98)00109-X.

Jie Chen, Shiguan Shan, Chu He, Guoying Zhao, Matti Pietikainen, Xilin Chen, y Wen Gao. WLD: A Robust Local Image Descriptor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1705–1720, Sept 2010. ISSN 0162-8828. DOI: 10.1109/TPAMI.2009.155.

Scott Shaobing Chen, David L. Donoho, y Michael A. Saunders. Atomic decomposition by basis pursuit. *SIAM Journal on Scientific Computing*, 20(1):33–61, December 1998. ISSN 1064-8275. DOI: 10.1137/S1064827596304010.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, y Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv e-prints*, arXiv:1512.01274, 2015a. URL <http://arxiv.org/abs/1512.01274>.

Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, y Yixin Chen. Compressing neural networks with the hashing trick. *arXiv e-prints*, arXiv:1504.04788, 2015b. URL <http://arxiv.org/abs/1504.04788>.

Daniel L. Chester. Why two hidden layers are better than one. En *ICNN'1990 Proceedings of the 4th IEEE Annual International Conference on Neural Networks*, volume 1, pages 265–268. Lawrence Erlbaum, January 1990.

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, y Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. En *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 571–582, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL <https://goo.gl/Tiys5K>.

KyungHyun Cho, Aaron C. Courville, y Yoshua Bengio. Describing multimedia content using attention-based encoder-decoder networks. *arXiv e-prints*, arXiv:1507.01053, 2015. URL <http://arxiv.org/abs/1507.01053>.

Sung-Bae Cho y Jin H. Kim. Rapid backpropagation learning algorithms. *Circuits, Systems and Signal Processing*, 12(2):155–175, Jun 1993. ISSN 1531-5878. DOI: 10.1007/BF01189872.

François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv e-prints*, arXiv:1610.02357, 2016. URL <http://arxiv.org/abs/1610.02357>.

Anna Choromanska, Mikael Henaff, Michaël Mathieu, Gérard Ben Arous, y Yann LeCun. The loss surface of multilayer networks. *AISTATS'2015*, arXiv:1412.0233, 2015. URL <http://arxiv.org/abs/1412.0233>.

Jan Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, KyungHyun Cho, y Yoshua Bengio. Attention-based models for speech recognition. *arXiv e-prints*, arXiv:1506.07503, 2015. URL <http://arxiv.org/abs/1506.07503>.

Bruce Christianson. Automatic Hessians by reverse accumulation. *IMA Journal of Numerical Analysis*, 12(2):135–150, 1992. DOI: 10.1093/imanum/12.2.135.

Joon Son Chung, Andrew W. Senior, Oriol Vinyals, y Andrew Zisserman. Lip reading sentences in the wild. *arXiv e-prints*, arXiv:1611.05358, 2016. URL <http://arxiv.org/abs/1611.05358>.

Patricia S. Churchland y Terrence J. Sejnowski. *The Computational Brain*. MIT Press, 1992. ISBN 0262031884.

Dan C. Ciresan, Ueli Meier, Luca Maria Gambardella, y Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12):3207–3220, 2010. ISSN 0899-7667. DOI: 10.1162/NECO_a_00052.

Dan C. Ciresan, Ueli Meier, Jonathan Masci, y Jürgen Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012a. ISSN 0893-6080. DOI: 10.1016/j.neunet.2012.02.023. Selected Papers from IJCNN 2011.

Dan C. Ciresan, Ueli Meier, y Jürgen Schmidhuber. Multi-column deep neural networks for image classification. En *CVPR'2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, 2012b. DOI: 10.1109/CVPR.2012.6248110.

Wesley A. Clark y Belmont G. Farley. Generalization of pattern recognition in a self-organizing system. En *Proceedings of the Western Joint Computer Conference, March 1-3, 1955*, AFIPS '55 (Western), pages 86–91, 1955. DOI: 10.1145/1455292.1455309.

Adam Coates y Andrew Y. Ng. The importance of encoding versus training with sparse coding and vector quantization. En *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 921–928. Omnipress, 2011. ISBN 978-1-4503-0619-5. URL http://www.icml-2011.org/papers/485_icmlpaper.pdf.

Adam Coates, Andrew Y. Ng, y Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. En *AISTATS'2011 Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 215–223, 2011. URL <https://goo.gl/Uyp1yZ>.

Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, y Bryan Catanzaro. Deep Learning with COTS HPC Systems. En Sanjoy Dasgupta y David McAllester, editores, *Proceedings of the 30th International Conference on International Conference on Machine Learning*, volume 28 of *ICML'13*, pages III.1337–III.1345, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/coates13.html>.

Michael Collins. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. En *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*, EMNLP '02, pages 1–8. Association for Computational Linguistics, 2002. DOI: 10.3115/1118693.1118694.

Michael Collins y Brian Roark. Incremental Parsing with the Perceptron Algorithm. En Donia Scott, Walter Daelemans, y Marilyn A. Walker, editores, *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics, 21–26 July, 2004, Barcelona, Spain.*, pages 111–118. Association for Computational Linguistics, 2004. URL <http://aclweb.org/anthology/P/P04/P04-1015.pdf>.

Ronan Collobert y Samy Bengio. Links between Perceptrons, MLPs and SVMs. En *Proceedings of the 21st International Conference on Machine Learning*, ICML '04, pages 23–, 2004. ISBN 1-58113-838-5. DOI: 10.1145/1015330.1015415.

Ronan Collobert y Jason Weston. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. En *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, 2008. ISBN 978-1-60558-205-4. DOI: 10.1145/1390156.1390177.

Ronan Collobert, Koray Kavukcuoglu, y Clément Farabet. Torch7: A Matlab-like Environment for Machine Learning. En *BigLearn, NIPS Workshop*, 2011a. URL http://ronan.collobert.com/pub/matos/2011_torch7_nipsw.pdf.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, y Pavel Kuksa. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, 12:2493–2537, November 2011b. ISSN 1532-4435. URL <http://www.jmlr.org/papers/v12/collobert11a.html>.

Andrew R. Conn, Nicholas I.M. Gould, y Philippe L. Toint. Convergence of quasi-Newton matrices generated by the symmetric rank one update. *Mathematical Programming*, 50(1):177–195, 1991. ISSN 1436-4646. DOI: 10.1007/BF01594934.

Andrew R. Conn, Katya Scheinberg, y Luis N. Vicente. *Introduction to Derivative-Free Optimization*. MPS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2009. ISBN 0898716683.

Corinna Cortes y Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995. ISSN 0885-6125. DOI: 10.1023/A:1022627411411.

Thomas M. Cover y Peter E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967. DOI: 10.1109/TIT.1967.1053964.

Nello Cristianini y John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000. ISBN 0521780195.

Xiaodong Cui, Vaibhava Goel, y Brian Kingsbury. Data augmentation for deep neural network acoustic modeling. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 23(9):1469–1477, September 2015. ISSN 2329-9290. DOI: 10.1109/TASLP.2015.2438544.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989. ISSN 0932-4194. DOI: 10.1007/BF02551274.

George E. Dahl, Dong Yu, Li Deng, y Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition.

IEEE Trans. Audio, Speech & Language Processing, 20(1):30–42, 2012.
 DOI: 10.1109/TASL.2011.2134090.

Ryan Dahl, Mohammad Norouzi, y Jonathon Shlens. Pixel recursive super resolution. *arXiv e-prints*, arXiv:1702.00783, 2017. URL <http://arxiv.org/abs/1702.00783>.

Jifeng Dai, Yi Li, Kaiming He, y Jian Sun. R-FCN: object detection via region-based fully convolutional networks. *arXiv e-prints*, arXiv:1605.06409, 2016. URL <http://arxiv.org/abs/1605.06409>.

Y.H. Dai y Y. Yuan. A nonlinear conjugate gradient method with a strong global convergence property. *SIAM Journal on Optimization*, 10(1):177–182, 1999. DOI: 10.1137/S1052623497318992.

Navneet Dalal y Bill Triggs. Histograms of oriented gradients for human detection. En *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893, 2005. DOI: 10.1109/CVPR.2005.177.

Christian Darken y John Moody. Note on learning rate schedules for stochastic optimization. En *Proceedings of the 3rd International Conference on Advances in Neural Information Processing Systems, NIPS'90*, pages 832–838. Morgan Kaufmann Publishers Inc., 1990. ISBN 1-55860-184-8. URL <https://goo.gl/HuC5Xu>.

Christian Darken y John Moody. Towards faster stochastic gradient search. En *Proceedings of the 4th International Conference on Neural Information Processing Systems, NIPS'91*, pages 1009–1016. Morgan Kaufmann Publishers Inc., 1991. ISBN 1-55860-222-4. URL <https://goo.gl/1Ypojb>.

Yann Dauphin, Razvan Pascanu, Çaglar Gülcöhre, Kyunghyun Cho, Surya Ganguli, y Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv e-prints*, arXiv:1406.2572, 2014a. URL <http://arxiv.org/abs/1406.2572>.

Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, y Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger, editores, *NIPS'2014 Advances in Neural Information Processing Systems 27*, pages 2933–2941. Curran Associates, Inc., 2014b. URL <https://goo.gl/dE72X5>.

Yann Dauphin, Harm de Vries, y Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization. En C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, y R. Garnett, editores, *NIPS'2015 Advances*

in Neural Information Processing Systems 28, pages 1504–1512. Curran Associates, Inc., 2015. URL <https://goo.gl/FJo8Sv>.

William C. Davidon. Variable metric method for minimization. *SIAM Journal on Optimization*, 1(1):1–17, 1991. DOI: 10.1137/0801001.

Peter Dayan y L.F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press, 2001. ISBN 0262041995.

Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, y Andrew Y. Ng. Large scale distributed deep networks. En *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pages 1223–1231. Curran Associates Inc., 2012. URL <https://goo.gl/tnZfdv>.

Olivier Delalleau y Yoshua Bengio. Shallow vs. Deep Sum-Product Networks. En J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *Advances in Neural Information Processing Systems 24*, pages 666–674. Curran Associates, Inc., 2011. URL <https:// goo.g1/qE6Thi>.

Arthur P. Dempster, Nan M. Laird, y Donald B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society (Series B: Methodological)*, 39(1):1–38, 1977. URL <http://www.jstor.org/stable/2984875>.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, y Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. En *CVPR'2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. DOI: 10.1109/CVPR.2009.5206848. URL http://www.image-net.org/papers/imagenet_cvpr09.pdf.

Li Deng y Dong Yu. *Deep Learning: Methods and Applications*. Foundations and Trends in Signal Processing. Now Publishers Inc., 2014. ISBN 1601988141.

Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, y Koray Kavukcuoglu. Natural neural networks. En Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, y Roman Garnett, editores, *NIPS'2015 Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2071–2079, 2015. URL <http://papers.nips.cc/paper/5953-natural-neural-networks>.

Laurent Dinh, Razvan Pascanu, Samy Bengio, y Yoshua Bengio. Sharp minima can generalize for deep nets. En Doina Precup y Yee Whye Teh, editores, *ICML'2017 Proceedings of the 34th International Conference on Machine Learning*, volume 70 de *Proceedings of Machine Learning Research*, páginas 1019–1028, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL <http://proceedings.mlr.press/v70/dinh17b.html>.

Pedro Domingos. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. Basic Books, 1st edition, 2015. ISBN 0465065708.

Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, y Trevor Darrell. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. *arXiv e-prints*, arXiv:1310.1531, 2013. URL <http://arxiv.org/abs/1310.1531>.

Chao Dong, Chen Change Loy, Kaiming He, , y Xiaoou Tang. Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2):295–307, 2016. ISSN 0162-8828. DOI: 10.1109/TPAMI.2015.2439281.

Keith L. Downing. *Intelligence Emerging: Adaptivity and Search in Evolving Neural Systems*. MIT Press, 2015. ISBN 0262029138.

Timothy Dozat. Incorporating Nesterov Momentum into Adam. *ICLR'2016 Workshop Track*, 2015. URL http://cs229.stanford.edu/proj2015/054_report.pdf.

G. P. Drago y S. Ridella. Statistically controlled activation weight initialization (SCAWI). *IEEE Transactions on Neural Networks*, 3(4):627–631, Jul 1992. ISSN 1045-9227. DOI: 10.1109/72.143378.

Stuart E. Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, 1962. DOI: 10.1016/0022-247X(62)90004-5.

Stuart E. Dreyfus. The computational solution of optimal control problems with time lag. *IEEE Transactions on Automatic Control*, 18(4):383–385, Aug 1973. ISSN 0018-9286. DOI: 10.1109/TAC.1973.1100330.

Stuart E. Dreyfus. Artificial neural networks, back propagation and the kelley-bryson gradient procedure. *Journal of Guidance, Control, and Dynamics*, 13(5):926–928, 1990. DOI: 10.2514/3.25422.

Harris Drucker y Yann LeCun. Double backpropagation increasing generalization performance. En *IJCNN'1991 International Joint Conference on Neural Networks, Seattle*, volume 2, páginas 145–150, Jul 1991. DOI: 10.1109/IJCNN.1991.155328.

Harris Drucker y Yann LeCun. Improving generalization performance using double backpropagation. *IEEE Transactions on Neural Networks*, 3(6):991–997, Nov 1992. ISSN 1045-9227. DOI: 10.1109/72.165600.

Ke-Lin Du y M.N.S. Swamy. *Neural Networks and Learning Machines*. Springer, 1st edition, 2014. ISBN 144715570X.

Włodzisław Duch. Uncertainty of data, fuzzy membership functions, and multilayer perceptrons. *IEEE Transactions on Neural Networks*, 16(1):10–23, Jan 2005. ISSN 1045-9227. DOI: 10.1109/TNN.2004.836200.

John Duchi, Elad Hazan, y Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, July 2011. ISSN 1532-4435. URL <http://jmlr.org/papers/v12/duchi11a.html>.

Richard O. Duda, Peter E. Hart, y David G. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000. ISBN 0471056693.

Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, y René Garcia. Incorporating second-order functional knowledge for better option pricing. En *Proceedings of the 13th International Conference on Neural Information Processing Systems*, NIPS’00, pages 451–457, Cambridge, MA, USA, 2000. MIT Press. URL <https://goo.gl/doB3GN>.

David Eagleman. *Incognito: The Secret Lives of the Brain*. Pantheon, 2011. ISBN 0307377334.

Bradley Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979. ISSN 0090-5364. DOI: 10.1214/aos/1176344552.

Agoston E. Eiben y James E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2nd edition, 2015. ISBN 3662448734.

Kihwan Eom, Kyungkwon Jung, y Harsha Sirisena. Performance improvement of backpropagation algorithm by automatic activation function gain tuning using fuzzy logic. *Neurocomputing*, 50:439–460, 2003. ISSN 0925-2312. DOI: 10.1016/S0925-2312(02)00576-3. URL <http://www.sciencedirect.com/science/article/pii/S0925231202005763>.

S. Ergezinger y E. Thomsen. An accelerated learning algorithm for multilayer perceptrons: optimization layer by layer. *IEEE Transactions on Neural Networks*, 6(1):31–42, Jan 1995. ISSN 1045-9227. DOI: 10.1109/72.363452.

Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, y Pascal Vincent. The difficulty of training deep architectures

and the effect of unsupervised pre-training. En David van Dyk y Max Welling, editores, *AISTATS'2009 Proceedings of the 12th International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 153–160, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR. URL <http://proceedings.mlr.press/v5/erhan09a.html>.

Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, y Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11:625–660, March 2010. ISSN 1532-4435. URL <http://www.jmlr.org/papers/v11/erhan10a.html>.

Alexander Fabisch, Yohannes Kassahun, Hendrik Wöhrle, y Frank Kirchner. Learning in compressed space. *Neural Networks*, 42:83–93, 2013. ISSN 0893-6080. DOI: 10.1016/j.neunet.2013.01.020.

Scott E. Fahlman. Faster-Learning Variations on Back-Propagation: An Empirical Study. En David S. Touretzky, Geoffrey E. Hinton, y Terrence J. Sejnowski, editores, *Proceedings of the 1988 Connectionist Models Summer School*, pages 38–51. Morgan Kaufmann, 1988. URL <https://www.cs.cmu.edu/~sef/sefPubs.htm>.

Scott E. Fahlman y Christian Lebiere. The cascade-correlation learning architecture. En D. S. Touretzky, editor, *NIPS'1989 Advances in Neural Information Processing Systems 2*, pages 524–532. Morgan-Kaufmann, 1990. URL <https://goo.gl/4tAUaK>.

Philipp Farber y Krste Asanovic. Parallel neural network training on Multi-Spert. En *Proceedings of 3rd International Conference on Algorithms and Architectures for Parallel Processing*, pages 659–666, 1997. DOI: 10.1109/ICAPP.1997.651531.

Belmont G. Farley y Wesley A. Clark. Simulation of self-organizing systems by digital computer. *Transactions of the IRE Professional Group on Information Theory*, 4(4):76–84, September 1954. ISSN 2168-2690. DOI: 10.1109/TIT.1954.1057468.

Donald W. Fausett. Strictly local backpropagation. En *IJCNN'1990 International Joint Conference on Neural Networks*, pages III:125–130, June 1990. DOI: 10.1109/IJCNN.1990.137834.

Laurene V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice Hall, 1994. ISBN 0133341860.

Usama Fayyad, Gregory Piatetsky-Shapiro, y Padhraic Smyth. The kdd process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, November 1996. ISSN 0001-0782. DOI: 10.1145/240455.240464.

Li Fei-Fei, R. Fergus, y P. Perona. One-shot learning of object categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):594–611, April 2006. ISSN 0162-8828. doi: 10.1109/TPAMI.2006.79.

Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, 2012. ISBN 1107422221.

Roger Fletcher. *Practical Methods of Optimization*. Wiley, 2nd edition, 2000. ISBN 0471915475.

Roger Fletcher y C. M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7(2):149–154, 1964. doi: 10.1093/comjnl/7.2.149.

Marcus Frean. The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2(2):198–209, April 1990. ISSN 0899-7667. doi: 10.1162/neco.1990.2.2.198.

James A. Freeman y David M. Skapura. *Neural Networks: Algorithms, Applications, and Programming Techniques*. Addison-Wesley, 1991. ISBN 0201513765.

James A. Freeman y David M. Skapura. *Redes neuronales: Algoritmos, aplicaciones y técnicas de programación*. Addison-Wesley / Díaz de Santos, 1993. ISBN 020160115X.

Yoav Freund y Robert E. Schapire. Experiments with a new boosting algorithm. En Lorenza Saitta, editor, *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*, pages 148–156. Morgan Kaufmann, 1996. ISBN 1-55860-419-7.

Yoav Freund y Robert E. Schapire. Large Margin Classification Using the Perceptron Algorithm. *Machine Learning*, 37(3):277–296, December 1999. ISSN 0885-6125. doi: 10.1023/A:1007662407062.

Jerome H. Friedman. On bias, variance, 0/1-loss, and the curse-of-dimensionality. *Data Mining and Knowledge Discovery*, 1(1):55–77, 1997. ISSN 1384-5810. doi: 10.1023/A:1009778005914.

Karl Pearson F.R.S. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine Series 6*, 2(11):559–572, 1901. doi: 10.1080/14786440109462720.

Kunihiro Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20(3):121–136, 1975. ISSN 1432-0770. doi: 10.1007/BF00342633.

Kunihiro Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980. ISSN 1432-0770. DOI: 10.1007/BF00344251.

Kunihiro Fukushima. A neural network for visual pattern recognition. *Computer*, 21(3):65–75, March 1988. ISSN 0018-9162. DOI: 10.1109/2.32.

Kunihiro Fukushima, Sei Miyake, y Takayuki Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):826–834, September/October 1983. ISSN 0018-9472. DOI: 10.1109/TSMC.1983.6313076.

A. Ronald Gallant y Halbert White. There exists a neural network that does not make avoidable mistakes. En *ICNN’1988 Proceedings of the IEEE 1988 International Conference on Neural Networks*, volume I, pages 657–664, July 1988. DOI: 10.1109/ICNN.1988.23903.

Stephen I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, June 1990. ISSN 1045-9227. DOI: 10.1109/72.80230.

Leon A. Gatys, Alexander S. Ecker, y Matthias Bethge. A neural algorithm of artistic style. *arXiv e-prints*, arXiv:1508.06576, 2015. URL <http://arxiv.org/abs/1508.06576>.

Leon A. Gatys, Matthias Bethge, Aaron Hertzmann, y Eli Shechtman. Preserving color in neural artistic style transfer. *arXiv e-prints*, arXiv:1606.05897, 2016a. URL <http://arxiv.org/abs/1606.05897>.

Leon A. Gatys, Alexander S. Ecker, y Matthias Bethge. Image style transfer using convolutional neural networks. En *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2414–2423, 2016b. ISBN 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.265.

Leon A. Gatys, Alexander S. Ecker, Matthias Bethge, Aaron Hertzmann, y Eli Shechtman. Controlling perceptual factors in neural style transfer. *arXiv e-prints*, arXiv:1611.07865, 2016c. URL <http://arxiv.org/abs/1611.07865>.

Leon A. Gatys, Alexander S. Ecker, Matthias Bethge, Aaron Hertzmann, y Eli Shechtman. Controlling perceptual factors in neural style transfer. En *CVPR’2017*, pages 3730–3738, 2017. ISBN 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.397.

Donald C. Gause y Gerald M. Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House, 1989. ISBN 0932633137.

Rong Ge, Furong Huang, Chi Jin, y Yang Yuan. Escaping From Saddle Points - Online Stochastic Gradient for Tensor Decomposition. *arXiv e-prints*, arxiv:1503.02101, 2015. URL <http://arxiv.org/abs/1503.02101>.

Stuart Geman, Elie Bienenstock, y René Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, January 1992. ISSN 0899-7667. DOI: 10.1162/neco.1992.4.1.1.

Rainer Gemulla, Erik Nijkamp, Peter J. Haas, y Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. En *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’11, pages 69–77, 2011. ISBN 978-1-4503-0813-7. DOI: 10.1145/2020408.2020426.

Claudio Gentile. A new approximate maximal margin classification algorithm. *Journal of Machine Learning Research*, 2:213–242, 2001. URL <http://www.jmlr.org/papers/v2/gentile01a.html>.

Wolfram Gerstner y Werner Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002. ISBN 0521890799.

Patrick R. Gill, Albert Wang, y Alyosha Molnar. The in-crowd algorithm for fast basis pursuit denoising. *IEEE Transactions on Signal Processing*, 59(10):4595–4605, Oct 2011. ISSN 1053-587X. DOI: 10.1109/TSP.2011.2161292.

Thomas Gilovich. *How We Know What Isn’t So*. Free Press, 1993. ISBN 0029117062.

Federico Girosi, Michael Jones, y Tomaso Poggio. Regularization theory and neural networks architectures. *Neural Computation*, 7(2):219–269, March 1995. ISSN 0899-7667. DOI: 10.1162/neco.1995.7.2.219.

Ross B. Girshick. Fast r-cnn. En *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015a. DOI: 10.1109/ICCV.2015.169.

Ross B. Girshick. Fast R-CNN. *arXiv e-prints*, arXiv:1504.08083, 2015b. URL <http://arxiv.org/abs/1504.08083>.

Ross B. Girshick, Jeff Donahue, Trevor Darrell, y Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CVPR’2014*, arXiv:1311.2524v5, 2014a. URL <http://arxiv.org/abs/1311.2524>.

Ross B. Girshick, Jeff Donahue, Trevor Darrell, y Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv e-prints*, abs/1311.2524v5, 2014b. URL <http://arxiv.org/abs/1311.2524>. Extended version of the CVPR'2014 paper.

Ross B. Girshick, Jeff Donahue, Trevor Darrell, y Jitendra Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(1):142–158, 2016. ISSN 0162-8828. DOI: 10.1109/TPAMI.2015.2437384.

Herbert Gish. A probabilistic approach to the understanding and training of neural network classifiers. En *ICASSP'1990 International Conference on Acoustics, Speech, and Signal Processing*, pages 1361–1364 vol.3, 1990. DOI: 10.1109/ICASSP.1990.115636.

James Gleick. *Chaos: Making a New Science*. Viking Books, 1987. ISBN 0670811785.

Xavier Glorot y Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. En Yee Whye Teh y D. Mike Titterington, editores, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256, 2010. URL <http://jmlr.org/proceedings/papers/v9/glorot10a>.

Xavier Glorot, Antoine Bordes, y Yoshua Bengio. Deep sparse rectifier neural networks. En Geoffrey Gordon, David Dunson, y Miroslav Dudík, editores, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *JMLR W&CP, Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL <http://proceedings.mlr.press/v15/glorot11a>.

Richard M. Golden. A unified framework for connectionist systems. *Biological Cybernetics*, 59(2):109–120, 1988. ISSN 1432-0770. DOI: 10.1007/BF00317773.

Faustino Gomez y Risto Mikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, January 1997. ISSN 1059-7123. DOI: 10.1177/105971239700500305.

Yunchao Gong, Liwei Wang, Ruiqi Guo, y Svetlana Lazebnik. Multi-scale orderless pooling of deep convolutional activation features. En David Fleet, Tomas Pajdla, Bernt Schiele, y Tinne Tuytelaars, editores,

Computer Vision – ECCV 2014, pages 392–407, 2014. ISBN 978-3-319-10584-0. doi: 10.1007/978-3-319-10584-0_26.

Ian Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, y Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. En *ICLR’2014 International Conference on Learning Representations*, 2014a. URL <https://arxiv.org/abs/1312.6082>.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, y Yoshua Bengio. Generative adversarial nets. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger, editores, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014b. URL <http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>.

Ian Goodfellow, Yoshua Bengio, y Aaron Courville. *Deep Learning*. MIT Press, 2016. ISBN 0262035618. URL <http://www.deeplearningbook.org>.

Ian J. Goodfellow. Technical report: Multidimensional, Downsampled Convolution for Autoencoders. Technical report, Université de Montréal, 2010.

Ian J. Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. *arXiv e-prints*, arXiv:1701.00160, 2016. URL <http://arxiv.org/abs/1701.00160>.

Ian J. Goodfellow, Aaron C. Courville, y Yoshua Bengio. Spike-and-slab sparse coding for unsupervised feature discovery. *NIPS’2011 Workshop on Challenges in Learning Hierarchical Models*, arXiv:1201.3382, 2011. URL <http://arxiv.org/abs/1201.3382>.

Ian J. Goodfellow, Aaron Courville, y Yoshua Bengio. Large-scale feature learning with spike-and-slab sparse coding. En *Proceedings of the 29th International Conference on International Conference on Machine Learning*, ICML’12, pages 1387–1394. Omnipress, 2012. ISBN 978-1-4503-1285-1. URL <http://icml.cc/2012/papers/718.pdf>.

Ian J. Goodfellow, Mehdi Mirza, Xia Da, Aaron C. Courville, y Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv e-prints*, 2013a. URL <http://arxiv.org/abs/1312.6211>.

Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, y Yoshua Bengio. Maxout networks. En *Proceedings of the 30th International Conference on Machine Learning, ICML 2013*,

Atlanta, GA, USA, 16-21 June 2013, pages 1319–1327, 2013b. URL <http://jmlr.org/proceedings/papers/v28/goodfellow13.html>.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, y Yoshua Bengio. Generative adversarial networks. *arXiv e-prints*, 2014c. URL <http://arxiv.org/abs/1406.2661>.

Ian J. Goodfellow, Jonathon Shlens, y Christian Szegedy. Explaining and harnessing adversarial examples. *ICLR'2015*, arXiv:1412.6572, 2015a. URL <http://arxiv.org/abs/1412.6572>.

Ian J. Goodfellow, Oriol Vinyals, y Andrew M. Saxe. Qualitatively characterizing neural network optimization problems. *ICLR'2015*, abs/1412.6544, 2015b. URL <http://arxiv.org/abs/1412.6544>.

Abigail Graese, Andras Rozsa, y Terrance E. Boult. Assessing threat of adversarial examples on deep neural networks. En *15th IEEE International Conference on Machine Learning and Applications, ICMLA 2016, Anaheim, CA, USA, December 18-20, 2016*, pages 69–74, 2016. ISBN 978-1-5090-6167-9. DOI: 10.1109/ICMLA.2016.0020.

Alex Graves. Practical variational inference for neural networks. En J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *NIPS'2011 Advances in Neural Information Processing Systems 24*, pages 2348–2356. Curran Associates, Inc., 2011. URL <https://goo.gl/MUKhLs>.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv e-prints*, arXiv:1308.0850, 2013. URL <http://arxiv.org/abs/1308.0850>.

Alex Graves, Greg Wayne, y Ivo Danihelka. Neural Turing Machines. *arXiv e-prints*, arXiv:1410.5401, 2014. URL <http://arxiv.org/abs/1410.5401>.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, y Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, October 2016. ISSN 0028-0836. DOI: 10.1038/nature20101.

Karol Gregor y Yann LeCun. Emergence of complex-like cells in a temporal product network with local receptive fields. *arXiv e-prints*, arXiv:1006.0448, 2010. URL <http://arxiv.org/abs/1006.0448>.

Andreas Griewank. Who Invented the Reverse Mode of Differentiation? *Documenta Mathematica*, Extra volume: Optimization Stories(ISMP):389–400, 2012. URL http://www.math.uiuc.edu/documenta/vol-ismp/52_griewank-andreas-b.html.

Andreas Griewank y Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Society for Industrial and Applied Mathematics, 2nd edition, 2008. ISBN 0898716594.

Charles G. Gross. Genealogy of the “grandmother cell”. *The Neuroscientist*, 8(5):512–518, 2002. DOI: 10.1177/107385802237175.

Stephen Grossberg. Pattern Learning by Functional-Differential Neural Networks with Arbitrary Path Weights. En Klaus Schmitt, editor, *Delay and Functional Differential Equations and their Applications*, pages 121–160. Academic Press, 1972. ISBN 978-0-12-627250-5. DOI: 10.1016/B978-0-12-627250-5.50009-5.

Stephen Grossberg. Contour Enhancement, Short Term Memory, and Constancies in Reverberating Neural Networks. *Studies in Applied Mathematics*, 52(3):213–257, 1973. ISSN 1467-9590. DOI: 10.1002/sapm1973523213.

Stephen Grossberg. *Pattern Learning by Functional-Differential Neural Networks with Arbitrary Path Weights*, pages 157–193. Springer, 1982a. ISBN 978-94-009-7758-7. DOI: 10.1007/978-94-009-7758-7_4.

Stephen Grossberg. *Contour Enhancement, Short Term Memory, and Constancies in Reverberating Neural Networks*, pages 332–378. Springer, 1982b. ISBN 978-94-009-7758-7. DOI: 10.1007/978-94-009-7758-7_8.

Frédéric Gruau. Genetic Micro Programming of Neural Networks. En Kenneth E. Kinnear, editor, *Advances in Genetic Programming*, pages 495–518. MIT Press, 1994. ISBN 0262111888.

Frédéric Gruau y L. Darrell Whitley. Adding Learning to the Cellular Development of Neural Networks: Evolution and the Baldwin Effect. *Evolutionary Computation*, 1(3):213–233, 1993. DOI: 10.1162/evco.1993.1.3.213.

Çaglar Gülcöhre y Yoshua Bengio. Knowledge Matters: Importance of Prior Information for Optimization. *ICLR’2013*, arXiv:1301.4083, 2013. URL <http://arxiv.org/abs/1301.4083>.

Çaglar Gülcöhre, Marcin Moczulski, y Yoshua Bengio. ADASECANT: robust adaptive secant method for stochastic gradient. *arXiv e-prints*, arXiv:1412.7419, 2014. URL <http://arxiv.org/abs/1412.7419>.

Caglar Gülcöhre, José Sotelo, Marcin Moczulski, y Yoshua Bengio. A robust adaptive stochastic gradient method for deep learning. En *IJCNN'2017 International Joint Conference on Neural Networks*, pages 125–132, May 2017. DOI: 10.1109/IJCNN.2017.7965845.

Amit Gupta y Siuwa M. Lam. Weight decay backpropagation for noisy data. *Neural Networks*, 11(6):1127 – 1138, 1998. ISSN 0893-6080. DOI: 10.1016/S0893-6080(98)00046-X.

Kevin Gurney. *An Introduction to Neural Networks*. Routledge, CRC Press, 1997. ISBN 1857286731.

Gaëtan Hadjeres y François Pachet. DeepBach: a Steerable Model for Bach chorales generation. *arXiv*, arXiv:1612.01010, 2016. URL <http://arxiv.org/abs/1612.01010>.

Martin T. Hagan, Howard B. Demuth, y Mark H. Beale. *Neural Network Design*. autoeditado, 1st edition, 2002. ISBN 0971732108.

Martin T. Hagan, Howard B. Demuth, Mark H. Beale, y Orlando de Jesús. *Neural Network Design*. autoeditado, 2nd edition, 2014. ISBN 0971732116.

Fredric M. Ham y Ivica Kostanic. *Principles of Neurocomputing for Science and Engineering*. McGraw-Hill, 2000. ISBN 0070259666.

Jiawei Han, Micheline Kamber, y Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 3rd edition, 2011. ISBN 0123814790.

J. M. Hannan y J. M. Bishop. A comparison of fast training algorithms over two real problems. En *Fifth International Conference on Artificial Neural Networks (Conf. Publ. No. 440)*, pages 1–6, Jul 1997. DOI: 10.1049/cp:19970692.

J. M. Hannan y J. Mark Bishop. A Class of Fast Artificial NN Training Algorithms. Technical Report JMH-JMB 01/96, Department of Cybernetics, University of Reading, UK, 1996.

Peter E. Hart, Nils J. Nilsson, y Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. ISSN 0536-1567. DOI: 10.1109/TSSC.1968.300136.

Peter E. Hart, Nils J. Nilsson, y Bertram Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bulletin*, (37):28–29, 1972. ISSN 0163-5719. DOI: 10.1145/1056777.1056779.

Ralph V.L. Hartley. Transmission of information. *Bell System Technical Journal*, 7(3):535–563, July 1928. ISSN 0005-8580. DOI: 10.1002/j.1538-7305.1928.tb01236.x.

Mohamad H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, 2003. ISBN 0262514672.

Jeff Hawkins. *On Intelligence*. Times Books, 2004. ISBN 0805074562.

Friedrich A. Hayek. *The Sensory Order: An Inquiry into the Foundations of Theoretical Psychology*. Routledge & Kegan Paul PLC, 1952. URL <https://archive.org/details/sensoryorderin00haye>.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. MacMillan Publishing Company, 1st edition, 1994. ISBN 0023527617.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2nd edition, 1998. ISBN 0132733501.

Simon Haykin. *Neural Networks and Learning Machines*. Pearson, 3rd edition, 2008. ISBN 0131471392.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv e-prints*, 2015a. URL <http://arxiv.org/abs/1502.01852>.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. Deep residual learning for image recognition. *arXiv e-prints*, arXiv:1512.03385, 2015b. URL <http://arxiv.org/abs/1512.03385>.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, y Jian Sun. Deep residual learning for image recognition. En *CVPR’2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. DOI: 10.1109/CVPR.2016.90.

Kaiming He, Georgia Gkioxari, Piotr Dollár, y Ross B. Girshick. Mask R-CNN. *arXiv e-prints*, arXiv:1703.06870, 2017. URL <http://arxiv.org/abs/1703.06870>.

Donald O. Hebb. *The Organization of Behavior*. John Wiley, New York, 1949.

Robert Hecht-Nielsen. Counterpropagation networks. *Applied Optics*, 26(23):4979–4984, 1987. DOI: 10.1364/AO.26.004979.

Robert Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, 1990. ISBN 0201093553.

Vishakh Hegde y Reza Zadeh. FusionNet: 3D Object Classification Using Multiple Data Representations. *arXiv e-prints*, arXiv:1607.05695, 2016. URL <http://arxiv.org/abs/1607.05695>.

John A. Hertz, Richard G. Palmer, y Anders Krogh. *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991. ISBN 0201503956.

Tom Heskes y Wim Wiegerinck. A theoretical comparison of batch-mode, on-line, cyclic, and almost-cyclic learning. *IEEE Transactions on Neural Networks*, 7(4):919–925, July 1996. ISSN 1045-9227. DOI: 10.1109/72.508935.

Magnus R. Hestenes y Eduard Stiefel. "Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952. DOI: 10.6028/jres.049.044.

Geoffrey Hinton. What's wrong with convolutional nets? *MIT, Brain & Cognitive Sciences - Fall Colloquium Series*, 2014a. URL <https://goo.gl/CS2Ugp>.

Geoffrey Hinton. AMA - Ask Me Anything. *reddit*, 2014b. URL <https://goo.gl/ldj45F>.

Geoffrey Hinton, Nitsh Srivastava, y Kevin Swersky. Neural networks for machine learning. *Coursera, MOOC video lectures*, 2012a.

Geoffrey Hinton, Oriol Vinyals, y Jeff Dean. Distilling the Knowledge in a Neural Network. *NIPS 2014 Deep Learning and Representation Learning Workshop*, arXiv:1503.02531, 2014a. URL <https://arxiv.org/abs/1503.02531>.

Geoffrey Hinton, Oriol Vinyals, y Jeff Dean. Dark knowledge. *TTIC Distinguished Lecture Series, Toyota Technological Institute at Chicago*, 2014b. URL <https://youtu.be/EK61htlw8hY>.

Geoffrey E. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 40(1):185–234, 1989. ISSN 0004-3702. DOI: 10.1016/0004-3702(89)90049-0.

Geoffrey E. Hinton y Steven J. Nowlan. How learning can guide evolution. *Complex Systems*, 1(3):495–502, 1987. ISSN 0891-2513. URL http://www.complex-systems.com/abstracts/v01_i03_a06.html.

Geoffrey E. Hinton y Sam T. Roweis. Stochastic neighbor embedding. En *NIPS'2002 Advances in Neural Information Processing Systems 15*, pages 857–864. MIT Press, 2002. URL <https://goo.gl/hkqjR3>.

Geoffrey E. Hinton y Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006. DOI: 10.1126/science.1127647.

Geoffrey E. Hinton y Drew van Camp. Keeping the neural networks simple by minimizing the description length of the weights. En *Proceedings of the 6th Annual Conference on Computational Learning Theory*, COLT '93, pages 5–13, 1993. ISBN 0-89791-611-5. DOI: 10.1145/168304.168306.

Geoffrey E. Hinton, James L. McClelland, y David E. Rumelhart. Distributed Representations. En David E. Rumelhart, James L. McClelland, y PDP Research Group, editores, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, pages 77–109. MIT Press, 1986. ISBN 026268053X.

Geoffrey E. Hinton, Simon Osindero, y Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006. DOI: 10.1162/neco.2006.18.7.1527.

Geoffrey E. Hinton, Alex Krizhevsky, y Sida D. Wang. Transforming auto-encoders. En Timo Honkela, Włodzisław Duch, Mark Girolami, y Samuel Kaski, editores, *Artificial Neural Networks and Machine Learning – ICANN 2011*, pages 44–51, 2011. ISBN 978-3-642-21735-7. DOI: 10.1007/978-3-642-21735-7_6.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, y Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv e-prints*, arXiv:1207.0580, 2012b. URL <http://arxiv.org/abs/1207.0580>.

Sepp Hochreiter y Juergen Schmidhuber. Simplifying neural nets by discovering flat minima. En G. Tesauro, D. S. Touretzky, y T. K. Leen, editores, *NIPS'1994 Advances in Neural Information Processing Systems 7*, pages 529–536. MIT Press, 1995. URL <https://goo.gl/1tc7xy>.

Sepp Hochreiter y Juergen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997. DOI: 10.1162/neco.1997.9.1.1.

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, y Jürgen Schmidhuber. Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies. En John F. Kolen y Stefan C. Kremer, editores, *A Field Guide to Dynamical Recurrent Neural Networks*, pages 237–244. IEEE Press, 2001. ISBN 0780353692.

Alan Lloyd Hodgkin y Andrew Fielding Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544,

1952. ISSN 0022-3751. DOI: 10.1113/jphysiol.1952.sp004764. URL <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413/>.

M. Hoehfeld y S. E. Fahlman. Learning with limited numerical precision using the cascade-correlation algorithm. *IEEE Transactions on Neural Networks*, 3(4):602–611, Jul 1992. ISSN 1045-9227. DOI: 10.1109/72.143374.

Douglas R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979. ISBN 0465026850.

John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982. URL <http://www.pnas.org/content/79/8/2554>.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. ISSN 0893-6080. DOI: 10.1016/0893-6080(91)90009-T.

Kurt Hornik, Maxwell Stinchcombe, y Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080. DOI: 10.1016/0893-6080(89)90020-8.

Kurt Hornik, Maxwell Stinchcombe, y Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5):551 – 560, 1990. ISSN 0893-6080. DOI: 10.1016/0893-6080(90)90005-6.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, y Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv e-prints*, arXiv:1704.04861, 2017. URL <http://arxiv.org/abs/1704.04861>.

Daniel Hsu, Nikos Karampatziakis, John Langford, y Alex J. Smola. *Parallel Online Learning*, pages 283–306. Cambridge University Press, 2012. ISBN 0521192242.

Gao Huang, Zhuang Liu, y Kilian Q. Weinberger. Densely connected convolutional networks. *arXiv e-prints*, arXiv:1608.06993, 2016. URL <http://arxiv.org/abs/1608.06993>.

Gary B. Huang, Manu Ramesh, Tamara Berg, y Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007. URL <http://vis-www.cs.umass.edu/lfw/lfw.pdf>.

Guang-Bin Huang, Qin-Yu Zhu, y Chee-Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1):489 – 501, 2006. ISSN 0925-2312. DOI: 10.1016/j.neucom.2005.12.126.

David H. Hubel. *Eye, Brain, and Vision*. W.H. Freeman, 1988. ISBN 0716750201. URL <http://hubel.med.harvard.edu/>.

Don R. Hush y Bill G. Horne. Progress in supervised neural networks. *IEEE Signal Processing Magazine*, 10(1):8–39, Jan 1993. ISSN 1053-5888. DOI: 10.1109/79.180705.

Frank Hutter, Holger H. Hoos, y Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. En Carlos A. Coello Coello, editor, *LION 5: 5th International Conference on Learning and Intelligent Optimization, Rome, Italy, January 17-21, 2011. Selected Papers*, pages 507–523. Springer, Berlin, Heidelberg, 2011. ISBN 978-3-642-25566-3. DOI: 10.1007/978-3-642-25566-3_40.

Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, y Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *arXiv e-prints*, arXiv:1602.07360, 2016. URL <http://arxiv.org/abs/1602.07360>.

Shun ichi Amari. Neural Learning in Structured Parameter Spaces - Natural Riemannian Gradient. En M. C. Mozer, M. I. Jordan, y T. Petsche, editores, *NIPS'1996 Advances in Neural Information Processing Systems 9*, pages 127–133. MIT Press, 1997. URL <https://goo.gl/y4jKF8>.

Ken ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2(3):183 – 192, 1989. ISSN 0893-6080. DOI: 10.1016/0893-6080(89)90003-8.

Christian Igel y Michael Hüsker. Empirical evaluation of the improved Rprop learning algorithms. *Neurocomputing*, 50:105 – 123, 2003. ISSN 0925-2312. DOI: 10.1016/S0925-2312(01)00700-7.

Harold Charles Daumé III. *Practical Structured Learning Techniques for Natural Language Processing*. PhD thesis, University of Southern California, 2006. URL <http://www.umiacs.umd.edu/~hal/docs/daume06thesis.pdf>.

W. Thomas Miller III, Richard S. Sutton, y Paul J. Werbos, editores. *Neural Networks for Control*. Neural Network Modeling and Connectionism. MIT Press, 1990. ISBN 0262132613.

Sergey Ioffe y Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. En Francis Bach y David Blei, editores, *ICML'2015 Proceedings of the 32nd International*

Conference on Machine Learning, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.
 URL <http://proceedings.mlr.press/v37/ioffe15.html>.

Masumi Ishikawa. Learning of modular structured networks. *Artificial Intelligence*, 75(1):51 – 62, 1995. ISSN 0004-3702. DOI: 10.1016/0004-3702(94)00061-5.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, y Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *arXiv e-prints*, arXiv:1611.07004, 2016. URL <http://arxiv.org/abs/1611.07004>.

Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, November 2003. ISSN 1045-9227. DOI: 10.1109/TNN.2003.820440.

Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295 – 307, 1988. ISSN 0893-6080. DOI: 10.1016/0893-6080(88)90003-2.

Max Jaderberg, Karen Simonyan, Andrew Zisserman, y Koray Kavukcuoglu. Spatial transformer networks. *arXiv e-prints*, arXiv:1506.02025, 2015. URL <http://arxiv.org/abs/1506.02025>.

Prateek Jain, Praneeth Netrapalli, y Sujay Sanghavi. Low-rank matrix completion using alternating minimization. En *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*, STOC’13, pages 665–674, 2013. ISBN 978-1-4503-2029-0. DOI: 10.1145/2488608.2488693.

Navdeep Jaitly y Geoffrey E. Hinton. Vocal Tract Length Perturbation (VTLN) Improves Speech Recognition. En *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013. URL <https://goo.gl/mMEiHM>.

Kevin Jarrett, Koray Kavukcuoglu, Marc’Aurelio Ranzato, y Yann LeCun. What is the best multi-stage architecture for object recognition? En *IEEE 12th International Conference on Computer Vision, ICCV 2009, Kyoto, Japan, September 27 - October 4, 2009*, pages 2146–2153, 2009. ISBN 978-1-4244-4420-5. DOI: 10.1109/ICCV.2009.5459469.

Yangqing Jia, Chang Huang, y Trevor Darrell. Beyond spatial pyramids: Receptive field learning for pooled image features. En *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3370–3377, 2012. DOI: 10.1109/CVPR.2012.6248076.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, y Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. En

Proceedings of the 22Nd ACM International Conference on Multimedia, MM '14, pages 675–678, 2014a. ISBN 978-1-4503-3063-3. DOI: 10.1145/2647868.2654889.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, y Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv e-prints*, arXiv:1408.5093, 2014b. URL <http://arxiv.org/abs/1408.5093>.

Kam-Chuen Jim, C.L. Giles, y Bill G. Horne. An analysis of noise in recurrent neural networks: convergence and generalization. *IEEE Transactions on Neural Networks*, 7(6):1424–1438, Nov 1996. ISSN 1045-9227. DOI: 10.1109/72.548170.

Aída Jiménez, Miguel Molina-Solana, Fernando Berzal, y Waldo Fajardo. Mining transposed motifs in music. *Journal of Intelligent Information Systems*, 36(1):99–115, 2011. DOI: 10.1007/s10844-010-0122-7.

Aída Jiménez, Fernando Berzal, y Juan Carlos Cubero. Using trees to mine multirelational databases. *Data Mining and Knowledge Discovery*, 24(1):1–39, 2012. DOI: 10.1007/s10618-011-0218-x.

E.M. Johansson, F.U. Dowla, y D.M. Goodman. Backpropagation learning for multilayer feed-forward neural networks using the conjugate gradient method. *International Journal of Neural Systems*, 02(04):291–301, 1991. DOI: 10.1142/S0129065791000261.

Andrew Johnson. *Spin-Images: A Representation for 3-D Surface Matching*. PhD thesis, Carnegie Mellon University, August 1997.

Justin Johnson, Alexandre Alahi, y Fei-Fei Li. Perceptual losses for real-time style transfer and super-resolution. *arXiv e-prints*, arXiv:1603.08155, 2016a. URL <http://arxiv.org/abs/1603.08155>.

Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yong-hui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda B. Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, y Jeffrey Dean. Google’s Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation. *arXiv e-prints*, abs/1611.04558, 2016b. URL <http://arxiv.org/abs/1611.04558>.

Bhautik J. Joshi, Kristen Stewart, y David Shapiro. Bringing impressionism to life with neural style transfer in come swim. *arXiv e-prints*, arXiv:1701.04928, 2017. URL <http://arxiv.org/abs/1701.04928>.

Rafal Jozefowicz, Wojciech Zaremba, y Ilya Sutskever. An empirical exploration of recurrent network architectures. En *Proceedings of the 32nd International Conference on International Conference on Machine*

Learning, volume 37 of *ICML'15*, pages 2342–2350. JMLR.org, 2015.
 URL <http://proceedings.mlr.press/v37/jozefowicz15.pdf>.

J. Stephen Judd. *Neural Network Design and the Complexity of Learning*.
 MIT Press, 1990. ISBN 0262100452.

Bela Julesz. Textons, the elements of texture perception, and their interactions. *Nature*, 290(5802):91–97, March 1981. ISSN 0028-0836.
 DOI: 10.1038/290091a0.

Daniel Jurafsky y James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice-Hall, 2nd edition, 2009. ISBN 0131873210.

Daniel Kahneman. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011. ISBN 0374275637.

Andrej Karpathy. CS231n: Convolutional Neural Networks for Visual Recognition. *Stanford*, 2017. URL <http://cs231n.github.io/>.

Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, y Li Fei-Fei. Large-scale video classification with convolutional neural networks. En *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014. DOI: 10.1109/CVPR.2014.223.

Koray Kavukcuoglu, Pierre Sermanet, Y-Lan Boureau, Karol Gregor, Michael Mathieu, y Yann L. Cun. Learning convolutional feature hierarchies for visual recognition. En *NIPS'2010 Advances in Neural Information Processing Systems 23*, pages 1090–1098, 2010. URL <https://goo.gl/1ZjGrU>.

Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, y Andrew Zisserman. The kinetics human action video dataset. *arXiv e-prints*, arXiv:1705.06950, 2017.
 URL <http://arxiv.org/abs/1705.06950>.

Henry J. Kelley. Gradient Theory of Optimal Flight Paths. *ARS Journal*, 30(10):947–954, 1960. DOI: 10.2514/8.5282.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, y Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *ICLR'2017*, abs/1609.04836, 2017. URL <http://arxiv.org/abs/1609.04836>.

Faisal Khan, Bilge Mutlu, y Xiaojin Zhu. How do humans teach: On curriculum learning and teaching dimension. En J. Shawe-Taylor,

R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *NIPS'2011 Advances in Neural Information Processing Systems 24*, pages 1449–1457. Curran Associates, Inc., 2011a. URL <https://goo.gl/c6TMn6>.

Gul Muhammad Khan, Julian F. Miller, y David M. Halliday. Evolution of Cartesian Genetic Programs for Development of Learning Neural Architecture. *Evolutionary Computation*, 19(3):469–523, 2011b. DOI: 10.1162/EVCO_a_00043.

Maryam Mahsal Khan, Arbab Masood Ahmad, Gul Muhammad Khan, y Julian F. Miller. Fast learning neural networks using Cartesian Genetic Programming. *Neurocomputing*, 121:274–289, 2013. DOI: 10.1016/j.neucom.2013.04.005.

Adnan Khashman. A modified backpropagation learning algorithm with added emotional coefficients. *IEEE Transactions on Neural Networks*, 19(11):1896–1909, Nov 2008. ISSN 1045-9227. DOI: 10.1109/TNN.2008.2002913.

Jack Kiefer y Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3): 462–466, 09 1952. DOI: 10.1214/aoms/1177729392.

Diederik P. Kingma y Jimmy Ba. Adam: A Method for Stochastic Optimization. *ICLR'2015*, arXiv:1412.6980, 2014. URL <http://arxiv.org/abs/1412.6980>.

Jon Kleinberg. An Impossibility Theorem for Clustering. En *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS'02, pages 463–470, 2002.

Christof Koch. *The Quest for Consciousness: A Neurobiological Approach*. Roberts & Company Publishers, 2004. ISBN 0974707708.

Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. En *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, 1995. ISBN 1-55860-363-8. URL <http://dl.acm.org/citation.cfm?id=1643031.1643047>.

Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982. ISSN 1432-0770. DOI: 10.1007/BF00337288.

Teuvo Kohonen. Learning Vector Quantization. En Michael A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 537–540. MIT Press, 1998. ISBN 0262511029.

R. Kozma, M. Sakuma, Y. Yokoyama, y M. Kitamura. On the accuracy of mapping by neural networks trained by backpropagation with forgetting. *Neurocomputing*, 13(2):295 – 311, 1996. ISSN 0925-2312. DOI: 10.1016/0925-2312(95)00094-1.

Werner Krauth y Marc Mezard. Learning algorithms with optimal stability in neural networks. *Journal of Physics A: Mathematical and General*, 20(11):L745, 1987. URL <http://stacks.iop.org/0305-4470/20/i=11/a=013>.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>.

Alex Krizhevsky. Convolutional deep belief networks on CIFAR-10. Technical report, University of Toronto, 2010. URL <https://www.cs.toronto.edu/~kriz/conv-cifar10-aug2010.pdf>.

Alex Krizhevsky, Ilya Sutskever, y Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. En *NIPS'2012 Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL <https://goo.gl/Ddt8Jb>.

J. K. Kruschke y J. R. Movellan. Benefits of gain: Speeded learning and minimal hidden layers in back-propagation networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(1):273–280, Jan 1991. ISSN 0018-9472. DOI: 10.1109/21.101159.

Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 50th anniversary edition edition, 1962. ISBN 0226458113.

Alexey Kurakin, Ian J. Goodfellow, y Samy Bengio. Adversarial examples in the physical world. *arXiv e-prints*, arXiv:1607.02533, 2016. URL <http://arxiv.org/abs/1607.02533>.

Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, July 1961. ISSN 0018-8646. DOI: 10.1147/rd.53.0183.

Kevin J. Lang y Geoffrey E. Hinton. The Development of the Time-Delay Neural Network Architecture for Speech Recognition. Technical Report CMU-CS-88-152, Department of Computer Science, Carnegie-Mellon University, 1988.

John Langford, Martin Zinkevich, y Alex J. Smola. Slow learners are fast. En Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, y A. Culotta, editores, *NIPS'2009 Advances in Neural Information Processing Systems 22*, pages 2331–2339. Curran Associates, Inc., 2009. URL <http://papers.nips.cc/paper/3888-slow-learners-are-fast.pdf>.

Pat Langley. *Elements of Machine Learning*. Morgan Kaufmann Publishers, 1995. ISBN 1558603018.

Hugo Larochelle y Yoshua Bengio. Classification using discriminative restricted boltzmann machines. En *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 536–543, 2008. ISBN 978-1-60558-205-4. DOI: 10.1145/1390156.1390224.

Hugo Larochelle, Dumitru Erhan, y Yoshua Bengio. Zero-data Learning of New Tasks. En *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 2 of *AAAI'08*, pages 646–651. AAAI Press, 2008. ISBN 978-1-57735-368-3. URL <http://dl.acm.org/citation.cfm?id=1620163.1620172>.

Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, y Pascal Lamblin. Exploring Strategies for Training Deep Neural Networks. *Journal of Machine Learning Research*, 10:1–40, June 2009. ISSN 1532-4435. URL <http://www.jmlr.org/papers/v10/larochelle09a.html>.

Julia A. Lasserre, Christopher M. Bishop, y T. P. Minka. Principled hybrids of generative and discriminative models. En *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 1, pages 87–94, June 2006. DOI: 10.1109/CVPR.2006.227.

Clifford Lau, editor. *Neural Networks - Theoretical Foundations and Analysis*. IEEE Press, 1992. ISBN 0879422807.

Svetlana Lazebnik, Cordelia Schmid, y Jean Ponce. Semi-Local Affine Parts for Object Recognition. En *British Machine Vision Conference*, 2004. URL <http://hal.archives-ouvertes.fr/docs/00/54/85/42/PDF/bmvc04.pdf>.

Quoc V. Le. Building high-level features using large scale unsupervised learning. En *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8595–8598, 2013. DOI: 10.1109/ICASSP.2013.6639343.

Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, y Andrew Y. Ng. On Optimization Methods for Deep Learning. En *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 265–272, USA, 2011. Omnipress. ISBN 978-1-4503-0619-5. URL http://www.icml-2011.org/papers/210_icmlpaper.pdf.

Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S. Corrado, Jeff Dean, y Andrew Y. Ng. Building high-level features using large scale unsupervised learning. En *Proceedings of the*

29th International Conference on International Conference on Machine Learning, ICML'12, pages 507–514, 2012. ISBN 978-1-4503-1285-1. URL <https://icml.cc/2012/papers/73.pdf>.

Yann LeCun. Une procédure d'apprentissage pour réseau à seuil asymétrique (a learning scheme for asymmetric threshold networks, in french). En *Proceedings of Cognitiva 85*, pages 599–604, Paris, France, 1985.

Yann LeCun. Learning process in an asymmetric threshold network. En E. Bienenstock, F. Fogelman-Soulie, y G. Weisbuch, editores, *Disordered Systems and Biological Organization*, volume 20 de *NATO ASI Series*, pages 233–240, Les Houches, France, 1986. Springer. ISBN 978-3-642-82657-3. DOI: 10.1007/978-3-642-82657-3_24.

Yann LeCun. Generalization and network design strategies. En R. Pfeifer, Z. Schreter, F. Fogelman, y L. Steels, editores, *Connectionism in perspective*. Elsevier, 1989a. ISBN 0444880615.

Yann LeCun. Generalization and network design strategies. Technical Report CRG-TR-89-4, Department of Computer Science, University of Toronto, 1989b.

Yann LeCun. Efficient Learning and Second-Order Methods. *NIPS 1993 Tutorial*, 1993. URL <https://goo.gl/njHrMC>.

Yann LeCun. AMA - Ask Me Anything. *reddit*, 2014. URL <https:// goo.gl/MK4H72>.

Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, R.E. Howard, Wayne E. Hubbard, y Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989a. ISSN 0899-7667. DOI: 10.1162/neco.1989.1.4.541.

Yann LeCun, Lawrence D. Jackel, Bernhard E. Boser, John S. Denker, , H.P. Graf, I. Guyon, Donnie Henderson, R.E. Howard, y Wayne E. Hubbard. Handwritten Digit Recognition: Applications of Neural Net Chips and Automatic Learning. *IEEE Communication*, pages 41–46, November 1989b. invited paper.

Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, R. E. Howard, Wayne E. Hubbard, y Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. En D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 396–404. Morgan-Kaufmann, 1990a. URL <https:// goo.gl/RKxWaW>.

Yann LeCun, Lawrence D. Jackel, Bernhard E. Boser, John S. Denker, , H.P. Graf, I. Guyon, Donnie Henderson, R.E. Howard, y Wayne E.

Hubbard. Handwritten Digit Recognition: Applications of Neural Net Chips and Automatic Learning. En Françoise Fogelman Soulié y Jeanny Hérault, editores, *Neurocomputing: Algorithms, Architectures and Applications*, pages 303–318. Springer Berlin Heidelberg, 1990b. ISBN 978-3-642-76153-9.

Yann LeCun, Patrice Y. Simard, y Barak Pearlmuter. Automatic Learning Rate Maximization by On-Line Estimation of the Hessian's Eigenvectors. En S. J. Hanson, J. D. Cowan, y C. L. Giles, editores, *Advances in Neural Information Processing Systems 5*, pages 156–163. Morgan-Kaufmann, 1993. URL <https://goo.gl/TW7uCr>.

Yann LeCun, Leon Bottou, Yoshua Bengio, y Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998a. ISSN 0018-9219. DOI: 10.1109/5.726791.

Yann LeCun, Leon Bottou, Genevieve B. Orr, y Klaus Robert Müller. Efficient BackProp. En Genevieve B. Orr y Klaus-Robert Müller, editores, *Neural Networks: Tricks of the Trade*, pages 9–50. Springer, 1998b. ISBN 3540653112. DOI: 10.1007/3-540-49430-8_2.

Yann LeCun, Fu Jie Huang, y Leon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. En *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 2, pages II–97–104, 2004. DOI: 10.1109/CVPR.2004.1315150.

Yann LeCun, Koray Kavukcuoglu, y Clément Farabet. Convolutional networks and applications in vision. En *ISCAS'2010 - Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 253–256, 2010. DOI: 10.1109/ISCAS.2010.5537907.

Yann LeCun, Léon Bottou, Genevieve B. Orr, y Klaus-Robert Müller. Efficient backprop. En Grégoire Montavon, Genevieve B. Orr, y Klaus-Robert Müller, editores, *Neural Networks: Tricks of the Trade*, pages 9–48. Springer, 2nd edition, 2012. ISBN 364235288X. DOI: 10.1007/978-3-642-35289-8_3.

Yann LeCun, Yoshua Bengio, y Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015. ISSN 0028-0836. DOI: 10.1038/nature14539.

Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew P. Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, y Wenzhe Shi. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv e-prints*, arXiv:1609.04802, 2016. URL <http://arxiv.org/abs/1609.04802>.

Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, y Zhuowen Tu. Deeply-Supervised Nets. *arXiv e-prints*, arXiv:1409.5185, 2014. URL <https://arxiv.org/abs/1409.5185>. Patent disclosure, UCSD Docket No. SD2014-313, filed on May 22, 2014.

Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, y Zhuowen Tu. Deeply-Supervised Nets. En Guy Lebanon y S. V. N. Vishwanathan, editores, *AISTATS'2015 Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pages 562–570, San Diego, California, USA, 09–12 May 2015. PMLR. URL <http://proceedings.mlr.press/v38/lee15a.html>.

Hahn-Ming Lee, Tzong-Ching Huang, y Chih-Ming Chen. Learning efficiency improvement of back propagation algorithm by error saturation prevention method. En *IJCNN'99 Proceedings of the IEEE International Joint Conference on Neural Networks*, volume 3, pages 1737–1742 vol.3, 1999. DOI: 10.1109/IJCNN.1999.832639.

Hahn-Ming Lee, Chih-Ming Chen, y Tzong-Ching Huang. Learning efficiency improvement of back-propagation algorithm by error saturation prevention method. *Neurocomputing*, 41(1):125 – 143, 2001. ISSN 0925-2312. DOI: 10.1016/S0925-2312(00)00352-0.

Honglak Lee, Roger Grosse, Rajesh Ranganath, y Andrew Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. En *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 609–616, 2009. ISBN 978-1-60558-516-1. DOI: 10.1145/1553374.1553453.

Y. Lee, S. H. Oh, y M. W. Kim. The effect of initial weights on premature saturation in back-propagation learning. En *IJCNN'91 Proceedings of the IEEE International Joint Conference on Neural Networks, Seattle, WA.*, volume 1, pages 765–770, Jul 1991. DOI: 10.1109/IJCNN.1991.155275.

Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, y Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861 – 867, 1993. ISSN 0893-6080. DOI: 10.1016/S0893-6080(05)80131-5.

Chi-Sing Leung, Kwok-Wo Wong, Pui-Fai Sum, y Lai-Wan Chan. A pruning method for the recursive least squared algorithm. *Neural Networks*, 14(2):147 – 174, 2001. ISSN 0893-6080. DOI: 10.1016/S0893-6080(00)00093-9.

Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):

164–168, 1944. ISSN 0033-569X. DOI: 10.1090/qam/10666. URL <http://www.jstor.org/stable/43633451>.

Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, y Bor-Yiing Su. Scaling distributed machine learning with the parameter server. En *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598. USENIX Association, 2014a. ISBN 978-1-931971-16-4. URL https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf.

Mu Li, Tong Zhang, Yuqiang Chen, y Alexander J. Smola. Efficient mini-batch training for stochastic optimization. En *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 661–670, 2014b. ISBN 978-1-4503-2956-9. DOI: 10.1145/2623330.2623612.

Min Lin, Qiang Chen, y Shuicheng Yan. Network in network. *arXiv e-prints*, arXiv:1312.4400, 2013. URL <http://arxiv.org/abs/1312.4400>.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, y C. Lawrence Zitnick. Microsoft coco: Common objects in context. En *ECCV'2014 European Conference on Computer Vision*, pages 740–755, 2014a. ISBN 978-3-319-10602-1. DOI: 10.1007/978-3-319-10602-1_48.

Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, y C. Lawrence Zitnick. Microsoft COCO: common objects in context. *arXiv*, arXiv:1405.0312, 2014b. URL <http://arxiv.org/abs/1405.0312>.

Greg Linden, Brent Smith, y Jeremy York. Amazon.Com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80, January 2003. ISSN 1089-7801. DOI: 10.1109/MIC.2003.1167344.

Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors (in Finnish). Master's thesis, University of Helsinki, Finland, 1970.

Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, Jun 1976. ISSN 1572-9125. DOI: 10.1007/BF01931367.

Richard P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(2):4–22, Apr 1987. ISSN 0740-7467. DOI: 10.1109/MASSP.1987.1165576.

Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, April 1988. ISSN 0885-6125. DOI: 10.1023/A:1022869011914.

Huan Liu, Farhad Hussain, Chew Lim Tan, y Manoranjan Dash. Discretization: An enabling technique. *Data Mining and Knowledge Discovery*, 6(4):393–423, 2002. ISSN 1573-756X. DOI: 10.1023/A:1016304305535.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, y Alexander C. Berg. SSD: single shot multibox detector. *arXiv e-prints*, arXiv:1512.02325, 2015. URL <http://arxiv.org/abs/1512.02325>.

Lennart Ljung. Analysis of recursive stochastic algorithms. *IEEE Transactions on Automatic Control*, 22(4):551–575, Aug 1977. ISSN 0018-9286. DOI: 10.1109/TAC.1977.1101561.

Shih-Chung B. Lo, Jyh-Shyan Lin, Matthew T. Freedman, y Seong Ki Mun. Computer-Assisted Diagnosis of Lung Nodule Detection using Artificial Convolution Neural Network. En *Proceedings of SPIE, Medical Imaging VII: Image Processing*, volume 1898, pages 859–869, 1993. DOI: 10.1117/12.154572.

Shih-Chung B. Lo, Heang-Ping Chan, Jyh-Shyan Lin, Huai Li, Matthew T. Freedman, y Seong K. Mun. Artificial Convolution Neural Network for Medical Image Pattern Recognition. *Neural Networks*, 8(7/8): 1201 – 1214, 1995a. ISSN 0893-6080. DOI: 10.1016/0893-6080(95)00061-5.

Shih-Chung B. Lo, S.L.A. Lou, Jyh-Shyan Lin, Matthew T. Freedman, M.V. Chien, y Seong Ki Mun. Artificial Convolution Neural Network Techniques and Application for Lung Nodule Detection. *IEEE Transactions on Medical Imaging*, 14(4):711–718, 1995b. ISSN 0278-0062. DOI: 10.1109/42.476112.

Shih-Chung B. Lo, Huai Li, Yue Wang, Lisa Kinnard, y Matthew T. Freedman. A Multiple Circular Path Convolution Neural Network System for Detection of Mammographic Masses. *IEEE Transactions on Medical Imaging*, 21(2):150–158, 2002. ISSN 0278-0062. DOI: 10.1109/42.993133.

Jonathan Long, Evan Shelhamer, y Trevor Darrell. Fully convolutional networks for semantic segmentation. *arXiv e-prints*, arXiv:1411.4038, 2014. URL <http://arxiv.org/abs/1411.4038>.

David G. Lowe. Object recognition from local scale-invariant features. En *Proceedings of the 7th IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157, 1999. DOI: 10.1109/ICCV.1999.790410.

David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. ISSN 1573-1405. DOI: 10.1023/B:VISI.0000029664.99615.94.

Fujun Luan, Sylvain Paris, Eli Shechtman, y Kavita Bala. Deep photo style transfer. *arXiv e-prints*, arXiv:1703.07511, 2017. URL <http://arxiv.org/abs/1703.07511>.

David G. Luenberger y Yinyu Ye. *Linear and Nonlinear Programming*. International Series in Operations Research & Management Science. Springer, 4th edition, 2016. ISBN 3319188410.

Andrew L. Maas, Awni Y. Hannun, y Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. En *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013. URL <https://goo.gl/5x3Cj6>.

David J. C. MacKay. Bayesian interpolation. *Neural Computation*, 4(3):415–447, May 1992. ISSN 0899-7667. DOI: 10.1162/neuro.1992.4.3.415.

Dougal Maclaurin, David K. Duvenaud, y Ryan P. Adams. Gradient-based hyperparameter optimization through reversible learning. *arXiv e-prints*, arXiv:1502.03492, 2015a. URL <http://arxiv.org/abs/1502.03492>.

Dougal Maclaurin, David K. Duvenaud, y Ryan P. Adams. Gradient-based hyperparameter optimization through reversible learning. En Francis R. Bach y David M. Blei, editores, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 de *JMLR Workshop and Conference Proceedings*, pages 2113–2122. JMLR.org, 2015b. URL <http://jmlr.org/proceedings/papers/v37/maclaurin15.html>.

William G. Macready y David H. Wolpert. What makes an optimization problem hard? *Complexity*, 1(5):40–46, 1996. DOI: 10.1002/cplx.6130010511.

George D. Magoulas, Michael N. Vrahatis, y George S. Androutsakis. Effective backpropagation training with variable stepsize. *Neural Networks*, 10(1):69 – 82, 1997. ISSN 0893-6080. DOI: 10.1016/S0893-6080(96)00052-4.

George D. Magoulas, Vassilis P. Plagianakos, y Michael N. Vrahatis. Globally convergent algorithms with local learning rates. *IEEE Transactions on Neural Networks*, 13(3):774–779, May 2002. ISSN 1045-9227. DOI: 10.1109/TNN.2002.1000148.

John Makhoul. Pattern recognition properties of neural networks. En *Proceedings of the 1991 IEEE Workshop on Neural Networks for Signal Processing*, pages 173–187, Sep 1991. DOI: 10.1109/NNSP.1991.239524.

Stephan Mandt, Matthew D. Hoffman, y David M. Blei. Stochastic gradient descent as approximate bayesian inference. *arXiv e-prints*, arXiv:1704.04289, 2017. URL <http://arxiv.org/abs/1704.04289>.

Marianthi Markatou, Hong Tian, Shameek Biswas, y George Hripcsak. Analysis of variance of cross-validation estimators of the generalization error. *Journal of Machine Learning Research*, 6:1127–1168, December 2005. ISSN 1532-4435. URL <http://www.jmlr.org/papers/volume6/markatou05a/markatou05a.pdf>.

Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963. DOI: 10.1137/0111030. URL <http://www.jstor.org/stable/2098941>.

David Marr. *Vision*. W.H. Freeman, 1983. ISBN 0716715678.

David Marr y Tomaso Poggio. Cooperative computation of stereo disparity. *Science*, 194(4262):283–287, 1976. ISSN 0036-8075. DOI: 10.1126/science.968482.

James Martens. Deep Learning via Hessian-free Optimization. En *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML’10, pages 735–742. Omnipress, 2010. ISBN 978-1-60558-907-7. URL <http://icml2010.haifa.il.ibm.com/papers/458.pdf>.

James Martens y Roger Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. En *Proceedings of the 32nd International Conference on Machine Learning, Volume 37*, ICML’15, pages 2408–2417. JMLR.org, 2015. URL <http://proceedings.mlr.press/v37/martens15.pdf>.

Jean-Pierre Martens. A stochastically motivated random initialization of pattern classifying MLPs. *Neural Processing Letters*, 3(1):23–29, 1996. ISSN 1573-773X. DOI: 10.1007/BF00417786.

Jean-Pierre Martens y Nico Weymaere. An equalized error backpropagation algorithm for the on-line training of multilayer perceptrons. *IEEE Transactions on Neural Networks*, 13(3):532–541, May 2002. ISSN 1045-9227. DOI: 10.1109/TNN.2002.1000122.

Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003. ISBN 0135974445.

Víctor Martínez, Fernando Berzal, y Juan Carlos Cubero. Adaptive degree penalization for link prediction. *Journal of Computational Science*, 13:1–9, 2016. DOI: 10.1016/j.jocs.2015.12.003.

Víctor Martínez, Fernando Berzal, y Juan Carlos Cubero. A survey of link prediction in complex networks. *ACM Computing Surveys*, 49(4):69:1–69:33, 2017. DOI: 10.1145/3012704.

Kiyotoshi Matsuoka y Jlanqiang Yi. Backpropagation based on the logarithmic error function and elimination of local minima. En *Proceedings of the 1991 IEEE International Joint Conference on Neural Networks*, pages 1117–1122 vol.2, 1991. DOI: 10.1109/IJCNN.1991.170546.

B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442 – 451, 1975. ISSN 0005-2795. DOI: 10.1016/0005-2795(75)90109-9.

Jürgen Mayer, Khaled Khairy, y Jonathon Howard. Drawing an elephant with four complex parameters. *American Journal of Physics*, 78(6):648–649, 2010. DOI: 10.1119/1.3254017.

Warren S. McCulloch y Walter Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943. ISSN 1522-9602. DOI: 10.1007/BF02478259.

Patrick McDaniel, Nicolas Papernot, y Z. Berkay Celik. Machine learning in adversarial settings. *IEEE Security & Privacy*, 14(3):68–72, 2016. ISSN 1540-7993. DOI: 10.1109/MSP.2016.51.

Sean McLoone y George W. Irwin. Fast parallel off-line training of multilayer perceptrons. *IEEE Transactions on Neural Networks*, 8(3):646–653, May 1997. ISSN 1045-9227. DOI: 10.1109/72.572103.

Sean McLoone, Michael D. Brown, George W. Irwin, y Gordon Lightbody. A hybrid linear/nonlinear training algorithm for feedforward neural networks. *IEEE Transactions on Neural Networks*, 9(4):669–684, Jul 1998. ISSN 1045-9227. DOI: 10.1109/72.701180.

Brendan McMahan y Matthew Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger, editores, *NIPS'2014 Advances in Neural Information Processing Systems 27*, pages 2915–2923. Curran Associates, Inc., 2014. URL <https://goo.gl/yQSu8D>.

H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel

Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnelsson, Tom Boulos, y Jeremy Kubica. Ad Click Prediction: A View from the Trenches. En *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1222–1230, 2013. ISBN 978-1-4503-2174-7. DOI: 10.1145/2487575.2488200.

Carver Mead. *Analog VLSI and Neural Systems*. Addison-Wesley, 1989. ISBN 0201059924.

Carver Mead y Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1979. ISBN 0201043580.

Bartlett W. Mel y Christof Koch. Sigma-Pi Learning: On Radial Basis Functions and Cortical Associative Learning. En David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 474–481. Morgan Kaufmann, 1990. ISBN 155860-1007. URL <https://goo.gl/EQ2i8G>.

Anil Menon, Kishan Mehrotra, Chilukuri K. Mohan, y Sanjay Ranka. Characterization of a class of sigmoid functions with application to neural networks. *Neural Networks*, 9(5):819–835, 1996. DOI: 10.1016/0893-6080(95)00107-7.

Murali M. Menon y Karl G. Heinemann. Classification of patterns using a self-organizing neural network. *Neural Networks*, 1(3):201 – 215, 1988. ISSN 0893-6080. DOI: 10.1016/0893-6080(88)90026-3. URL <http://www.sciencedirect.com/science/article/pii/0893608088900263>.

Grégoire Mesnil, Yann Dauphin, Xavier Glorot, Salah Rifai, Yoshua Bengio, Ian Goodfellow, Erick Lavoie, Xavier Muller, Guillaume Desjardins, David Warde-Farley, Pascal Vincent, Aaron Courville, y James Bergstra. Unsupervised and transfer learning challenge: a deep learning approach. En Isabelle Guyon, Gideon Dror, Vincent Lemaire, Graham Taylor, y Daniel Silver, editores, *Proceedings of ICML'2011 Workshop on Unsupervised and Transfer Learning*, volume 27 de *Proceedings of Machine Learning Research*, pages 97–110, Bellevue, Washington, USA, 02 Jul 2011. PMLR. URL <http://proceedings.mlr.press/v27/mesnil12a.html>.

Marc Mézard y Jean-Pierre Nadal. Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A: Mathematical and General*, 22(12):2191, 1989. URL <http://stacks.iop.org/0305-4470/22/i=12/a=019>.

Krystian Mikolajczyk y Cordelia Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(10):1615–1630, 2005. ISSN 0162-8828. DOI: 10.1109/TPAMI.2005.188.

Tomas Mikolov, Kai Chen, Greg Corrado, y Jeffrey Dean. Efficient estimation of word representations in vector space. *ICLR'2013*, arXiv:1301.3781, 2013a. URL <http://arxiv.org/abs/1301.3781>.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, y Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *arXiv e-prints*, arXiv:1310.4546, 2013b. URL <http://arxiv.org/abs/1310.4546>.

George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, March 1956. DOI: 10.1037/h0043158.

Julian F. Miller y Gul Muhammad Khan. Where is the brain inside the brain? on why artificial neural networks should be developmental. *Memetic Computing*, 3(3):217–228, 2011. DOI: 10.1007/s12293-011-0062-y.

Julian F. Miller, Dominic Job, y Vesselin K. Vassilev. Principles in the Evolutionary Design of Digital Circuits - Part I. *Genetic Programming and Evolvable Machines*, 1(1-2):7–35, April 2000a. ISSN 1389-2576. DOI: 10.1023/A:1010016313373.

Julian F. Miller, Dominic Job, y Vesselin K. Vassilev. Principles in the Evolutionary Design of Digital Circuits - Part II. *Genetic Programming and Evolvable Machines*, 1(3):259–288, July 2000b. ISSN 1389-2576. DOI: 10.1023/A:1010066330916.

Brenda Milner. Clues to the cerebral organization of memory. En Pierre A. Buser y Arlette Rougeul-Buser, editores, *Cerebral Correlates of Conscious Experience*, pages 139–153. Elsevier, 1978. ISBN 0720406595.

A.A. Minai y R.D. Williams. Back-propagation heuristics: A study of the extended delta-bar-delta algorithm. En *1990 IJCNN International Joint Conference on Neural Networks*, pages 595–600 vol.1, June 1990. DOI: 10.1109/IJCNN.1990.137634.

Marvin Minsky. *Theory of Neural-Analog Reinforcement Systems and Its Application to the Brain-Model Problem*. PhD thesis, Princeton University, 1954.

Marvin Minsky y Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1st edition, 1969. ISBN 0262130432.

Marvin Minsky y Seymour Papert. Artificial Intelligence: Progress Report. *MIT*, Artificial Intelligence Memo No. 252, 1972. URL <https://goo.gl/NH6vu5>.

Marvin Minsky y Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, expanded edition, 1987. ISBN 0262631113.

Dmytro Mishkin y Jiri Matas. All you need is a good init. *ICLR'2016*, arXiv:1511.06422, 2016. URL <http://arxiv.org/abs/1511.06422>.

Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, 1st edition, 1997. ISBN 0070428077.

Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, y Shin Ishii. Virtual Adversarial Training: a Regularization Method for Supervised and Semi-supervised Learning. *arXiv e-prints*, arXiv:1704.03976, 2016a. URL <http://arxiv.org/abs/1704.03976>.

Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, Ken Nakae, y Shin Ishii. Distributional Smoothing by Virtual Adversarial Examples. *ICLR'2016*, arXiv:1507.00677, 2016b. URL <http://arxiv.org/abs/1507.00677>.

Takeru Miyato, Andrew M. Dai, y Ian J. Goodfellow. Adversarial Training Methods for Semi-Supervised Text Classification. *ICLR'2017*, arXiv:1605.07725, 2017. URL <http://arxiv.org/abs/1605.07725>.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, y Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. DOI: 10.1038/nature14236.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, y Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv e-prints*, arXiv:1602.01783, 2016.

Cleve Moler y Michael T. Heath, editores. *Matrix Computation on Distributed Memory Multiprocessors*. Society for Industrial and Applied Mathematics, 1986. ISBN 0898712092.

Martin Fodslette Moller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6(4):525 – 533, 1993. ISSN 0893-6080. DOI: 10.1016/S0893-6080(05)80056-5.

Grégoire Montavon, Genevieve B. Orr, y Klaus-Robert Müller, editores. *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700 of *Lecture Notes in Computer Science*. Springer, 2012. ISBN 364235288X. doi: 10.1007/978-3-642-35289-8.

Guido F. Montúfar. Universal approximation depth and errors of narrow belief networks with discrete units. *Neural Computation*, 26(7):1386–1407, July 2014. ISSN 0899-7667. doi: 10.1162/NECO_a_00601.

Guido F. Montúfar, Razvan Pascanu, Kyunghyun Cho, y Yoshua Bengio. On the number of linear regions of deep neural networks. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger, editores, *NIPS'2014 Advances in Neural Information Processing Systems 27*, pages 2924–2932. Curran Associates, Inc., 2014. URL <https://goo.gl/tGAZ46>.

Cristopher Moore y Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011. ISBN 0199233217.

Konda Reddy Mopuri, Utsav Garg, y R. Venkatesh Babu. CNN fixations: An unraveling approach to visualize the discriminative image regions. *arXiv e-prints*, arXiv:1708.06670, 2017. URL <http://arxiv.org/abs/1708.06670>.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, y Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017. ISSN 0036-8075. doi: 10.1126/science.aam6960.

Alexander Mordvintsev, Christopher Olah, y Mike Tyka. Inceptionism: Going Deeper into Neural Networks. Technical report, Google Research Blog, 2015. URL <https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html>.

Jorge J. Moré. The Levenberg–Marquardt algorithm: Implementation and theory. En G. A. Watson, editor, *Numerical Analysis: Proceedings of the Biennial Conference Held at Dundee, June 28–July 1, 1977*, volume 630 of *Lecture Notes in Mathematics*, pages 105–116. Springer, 1977. ISBN 978-3-540-35972-2. doi: 10.1007/BFb0067700.

David E. Moriarty y Risto Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5(4):373–399, December 1997. ISSN 1063-6560. doi: 10.1162/evco.1997.5.4.373.

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012. ISBN 0262018020.

Vinod Nair y Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. En Johannes Fürnkranz y Thorsten Joachims, editores, *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814. Omnipress, 2010. ISBN 978-1-60558-907-7. URL <https://www.cs.toronto.edu/~hinton/absps/reluICML.pdf>.

Takéhiko Nakama. Theoretical analysis of batch and on-line training for gradient descent learning in neural networks. *Neurocomputing*, 73(1):151 – 159, 2009. ISSN 0925-2312. DOI: 10.1016/j.neucom.2009.05.017.

Sridhar Narayan. The generalized sigmoid activation function: Competitive supervised learning. *Information Sciences*, 99(1):69 – 82, 1997. ISSN 0020-0255. DOI: 10.1016/S0020-0255(96)00200-9.

Stephen G. Nash. A survey of truncated-Newton methods. *Journal of Computational and Applied Mathematics*, 124(1):45 – 59, 2000. ISSN 0377-0427. DOI: 10.1016/S0377-0427(00)00426-X. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

B. K. Natarajan. Sparse approximate solutions to linear systems. *SIAM Journal on Computing*, 24(2):227–234, 1995. DOI: 10.1137/S0097539792240406.

Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, y James Martens. Adding gradient noise improves learning for very deep networks. *arXiv e-prints*, arXiv:1511.06807, 2015. URL <http://arxiv.org/abs/1511.06807>.

John A. Nelder y Roger Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965. DOI: 10.1093/comjnl/7.4.308.

Arkadi Nemirovski. Numerical Methods for Nonlinear Continuous Optimization. Technical report, Technion, Israel Institute of Technology, 1999. URL http://www2.isye.gatech.edu/~nemirovs/Lect_OptII.pdf.

Yurii Nesterov. A method of solving a convex programming problem with convergence rate $O(1/\text{sqr}(k))$. *Soviet Mathematics Doklady*, 27: 372–376, 1983.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, y Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. En *NIPS'2011 Workshop on Deep Learning and Unsupervised Feature Learning*, 2011. URL http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

Allen Newell, John C. Shaw, y Herbert A. Simon. Report on a General Problem-Solving Program. En *Proceedings of the International Conference on Information Processing*, pages 256–264, 1959.

Andrew Yan-Tak Ng. Advice for Applying Machine Learning. *Stanford CS229 Machine Learning*, 2015. URL <http://cs229.stanford.edu/materials/ML-advice.pdf>.

S.C. Ng, S.H. Leung, y A. Luk. Fast convergent generalized back-propagation algorithm with constant learning rate. *Neural Processing Letters*, 9(1):13–23, Feb 1999. ISSN 1573-773X. DOI: 10.1023/A:1018611626332.

Jiquan Ngiam, Zhenghao Chen, Daniel Chia, Pang W. Koh, Quoc V. Le, y Andrew Y. Ng. Tiled convolutional neural networks. En J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, y A. Culotta, editores, *NIPS'2010 Advances in Neural Information Processing Systems 23*, pages 1279–1287. Curran Associates, Inc., 2010. URL <https://goo.gl/DoQGsz>.

Anh Nguyen, Jason Yosinski, Yoshua Bengio, Alexey Dosovitskiy, y Jeff Clune. Plug & play generative networks: Conditional iterative generation of images in latent space. *arXiv e-prints*, arXiv:1612.00005, 2016. URL <http://arxiv.org/abs/1612.00005>.

Anh Nguyen, Jeff Clune, Yoshua Bengio, Alexey Dosovitskiy, y Jason Yosinski. Plug & play generative networks: Conditional iterative generation of images in latent space. En *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 3510–3520, 2017. ISBN 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.374.

Derrick Nguyen y Bernard Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. En *IJCNN'1990 International Joint Conference on Neural Networks, Washington DC*, pages III:21–26, June 1990. DOI: 10.1109/IJCNN.1990.137819.

Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL <http://neuralnetworksanddeeplearning.com/>.

Robert H. Nielsen. Theory of the Backpropagation Neural Network. En *IJCNN'1989, Proceedings of the International Joint Conference on Neural Networks*, (Washington, DC), volume I, pages 593–605, 1989. DOI: 10.1109/IJCNN.1989.118638.

Jorge Nocedal. Updating Quasi-Newton Matrices with Limited Storage. *Mathematics of Computation*, 35(151):773–782, 1980. ISSN 0025-5718.

DOI: 10.1090/S0025-5718-1980-0572855-7. URL <https://www.jstor.org/stable/2006193>.

Jorge Nocedal y Stephen Wright. *Numerical Optimization*. Springer, 2nd edition, 2006. ISBN 0387303030.

Albert B.J. Novikoff. On convergence proofs on perceptrons. En *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622, 1962.

Steven J. Nowlan y Geoffrey E. Hinton. Adaptive Soft Weight Tying using Gaussian Mixtures. En J. E. Moody, S. J. Hanson, y R. P. Lippmann, editores, *NIPS'1991 Advances in Neural Information Processing Systems 4*, pages 993–1000. Morgan-Kaufmann, 1991. URL <https://goo.gl/4J9onY>.

Steven J. Nowlan y Geoffrey E. Hinton. Simplifying Neural Networks by Soft Weight-sharing. *Neural Computation*, 4(4):473–493, July 1992. ISSN 0899-7667. DOI: 10.1162/neco.1992.4.4.473.

Sang-Hoon Oh. Improving the error backpropagation algorithm with a modified error function. *IEEE Transactions on Neural Networks*, 8(3):799–803, May 1997. ISSN 1045-9227. DOI: 10.1109/72.572117.

Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, 15(3):267–273, 1982. ISSN 1432-1416. DOI: 10.1007/BF00275687.

Chris Olah, Alexander Mordvintsev, y Ludwig Schubert. Feature visualization. *Distill*, 2017. DOI: 10.23915/distill.00007. URL <https://distill.pub/2017/feature-visualization>.

Maxime Oquab, Leon Bottou, Ivan Laptev, y Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. En *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1717–1724, June 2014. DOI: 10.1109/CVPR.2014.222.

Mark Palatucci, Dean Pomerleau, Geoffrey E Hinton, y Tom M. Mitchell. Zero-shot learning with semantic output codes. En *NIPS'2009 Advances in Neural Information Processing Systems 22*, pages 1410–1418. Curran Associates, Inc., 2009. URL <https://goo.gl/K7uCfH>.

Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, y Ananthram Swami. The limitations of deep learning in adversarial settings. En *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 372–387, 2016. ISBN 978-1-5090-1751-5. DOI: 10.1109/EuroSP.2016.36.

Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, y Ananthram Swami. Practical black-box attacks against machine learning. En *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 506–519, 2017. ISBN 978-1-4503-4944-4. DOI: 10.1145/3052973.3053009.

David B. Parker. Learning-Logic: Casting the Cortex of the Human Brain in Silicon. Technical Report TR-47, Center for Computational Research in Economics and Management Science, Alfred P. Sloan School of Management, MIT, 1985.

A. G. Parlos, B. Fernandez, A. F. Atiya, J. Muthusami, y W. K. Tsai. An accelerated learning algorithm for multilayer perceptron networks. *IEEE Transactions on Neural Networks*, 5(3):493–497, May 1994. ISSN 1045-9227. DOI: 10.1109/72.286921.

Razvan Pascanu y Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv e-prints*, arXiv:1301.3584, 2014. URL <http://arxiv.org/abs/1301.3584>.

Razvan Pascanu, Tomas Mikolov, y Yoshua Bengio. On the difficulty of training recurrent neural networks. En *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III.1310–1318. JMLR.org, 2013. URL <http://proceedings.mlr.press/v28/pascanu13.pdf>.

Razvan Pascanu, Yann N. Dauphin, Surya Ganguli, y Yoshua Bengio. On the saddle point problem for non-convex optimization. *arXiv e-prints*, arXiv:1405.4604, 2014. URL <http://arxiv.org/abs/1405.4604>.

Y. C. Pati, R. Rezaifar, y P. S. Krishnaprasad. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. En *Proceedings of 27th Asilomar Conference on Signals, Systems and Computers*, pages 40–44 vol.1, Nov 1993. DOI: 10.1109/ACSSC.1993.342465.

Romain Paulus, Caiming Xiong, y Richard Socher. A deep reinforced model for abstractive summarization. *arXiv e-prints*, arXiv:1705.04304, 2017. URL <http://arxiv.org/abs/1705.04304>.

Claudia Pérez-D'Arpino y Julie A. Shah. C-LEARN: Learning Geometric Constraints from Demonstrations for Multi-Step Manipulation in Shared Autonomy. En *Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA'2017)*, pages 4058–4065, 2017. ISBN 978-1-5090-4633-1. DOI: 10.1109/ICRA.2017.7989466.

Marcus Pfister y Raúl Rojas. Hybrid learning algorithms for feed-forward neural networks. En Bernd Reusch, editor, *Fuzzy Logik: Theorie und Praxis 4. Dortmunder Fuzzy-Tage Dortmund, 6.–8. Juni 1994*, pages 61–68, 1994. ISBN 978-3-642-79386-8. doi: 10.1007/978-3-642-79386-8_8.

P. K. H. Phua y Daohua Ming. Parallel nonlinear optimization techniques for training neural networks. *IEEE Transactions on Neural Networks*, 14(6):1460–1468, Nov 2003. ISSN 1045-9227. doi: 10.1109/TNN.2003.820670.

David C. Plaut y Geoffrey E. Hinton. Learning sets of filters using back-propagation. *Computer Speech and Language*, 2(1):35–61, 1987. ISSN 0885-2308. doi: 10.1016/0885-2308(87)90026-X.

Tomaso Poggio y Federico Girosi. Networks for approximation and learning. *Proceedings of the IEEE*, 78(9):1481–1497, Sep 1990. ISSN 0018-9219. doi: 10.1109/5.58326.

Elijah Polak y G. Ribi  re. Note sur la convergence de m  thodes de directions conjugu  es. *Revue fran  aise d'onformatique et de recherche op  rationnelle. S  rie rouge*, 3(R1):35–43, 1969. URL http://www.numdam.org/article/M2AN_1969__3_1_35_0.pdf.

Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1 – 17, 1964. ISSN 0041-5553. doi: 10.1016/0041-5553(64)90137-5.

Boris T. Polyak. New method of stochastic approximation. *Automation and Remote Control (translated from Russian)*, 51(7.2):937–946, 1990. URL <https://goo.gl/CzEdSH>.

Boris T. Polyak y A. B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, July 1992. ISSN 0363-0129. doi: 10.1137/0330046.

Ben Poole, Jascha Sohl-Dickstein, y Surya Ganguli. Analyzing noise in autoencoders and deep networks. *arXiv e-prints*, arXiv:1406.1831, 2014. URL <http://arxiv.org/abs/1406.1831>.

Javier Portilla y Eero P. Simoncelli. A parametric texture model based on joint statistics of complex wavelet coefficients. *International Journal of Computer Vision*, 40(1):49–70, 2000. ISSN 1573-1405. doi: 10.1023/A:1026553619983.

M. J. D. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal*, 7(2):155–162, 1964. doi: 10.1093/comjnl/7.2.155.

M. J. D. Powell. Restart procedures for the conjugate gradient method. *Mathematical Programming*, 12(1):241–254, Dec 1977. ISSN 1436-4646. DOI: 10.1007/BF01593790.

William H. Press, Saul A. Teukolsky, William T. Vetterling, y Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007. ISBN 0521880688.

Roger S. Pressman. *Ingeniería del Software: Un enfoque práctico*. McGraw-Hill Interamericana, 1995. ISBN 8448100263.

Jonathan K. Pritchard, Matthew Stephens, y Peter Donnelly. Inference of population structure using multilocus genotype data. *Genetics*, 155(2):945–959, 2000. ISSN 0016-6731. URL <http://www.genetics.org/content/155/2/945>.

Dale Purves. *Brains: How they seem to work and what that tells us about who we are*. FT Press Science, Pearson, 2010. ISBN 0137055099.

Dale Purves, George J. Augustine, David Fitzpatrick, William C. Hall, Anthony-Samuel LaMantia, James O. McNamara, y S. Mark Williams. *Neuroscience*. Sinauer Associates, 3rd edition, 2004. ISBN 0878937250.

Dale Purves, Elizabeth M. Brannon, Roberto Cabeza, Scott A. Huettel, Kevin S. LaBar, Michael L. Platt, y Marty G. Woldorff. *Principles of Cognitive Neuroscience*. Sinauer Associates, 1st edition, 2007. ISBN 0878936947.

Alec Radford, Luke Metz, y Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *ICLR'2016*, arXiv:1511.06434, 2016. URL <http://arxiv.org/abs/1511.06434>.

Marc'Aurelio Ranzato, Christopher S. Poultney, Sumit Chopra, y Yann LeCun. Efficient learning of sparse representations with an energy-based model. En *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 1137–1144, 2006. URL <https://goo.gl/pKBzV0>.

Marc'Aurelio Ranzato, Fu Jie Huang, Y-Lan Boureau, y Yann LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. En *CVPR'2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007. DOI: 10.1109/CVPR.2007.383157.

Antti Rasmus, Harri Valpola, Mikko Honkala, Mathias Berglund, y Tapani Raiko. Semi-supervised learning with ladder network. *arXiv*

e-prints, arXiv:1507.02672, 2015. URL <http://arxiv.org/abs/1507.02672>.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, y Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. En *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 525–542. Springer, 2016a. ISBN 978-3-319-46492-3. DOI: 10.1007/978-3-319-46493-0_32.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, y Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *arXiv*, arXiv:1603.05279, 2016b. URL <http://arxiv.org/abs/1603.05279>.

Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, y Stefan Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. *arXiv e-prints*, arXiv:1403.6382, 2014. URL <http://arxiv.org/abs/1403.6382>.

Benjamin Recht, Christopher Re, Stephen Wright, y Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. En J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, y K. Q. Weinberger, editores, *NIPS'2011 Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011. URL <https://goo.gl/qYqnQ3>.

Joseph Redmon y Ali Farhadi. YOLO9000: better, faster, stronger. *arXiv e-prints*, arXiv:1612.08242, 2016. URL <http://arxiv.org/abs/1612.08242>.

Joseph Redmon y Ali Farhadi. YOLO9000: better, faster, stronger. En *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6517–6525, 2017. ISBN 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.690.

Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, y Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *arXiv e-prints*, arXiv:1506.02640, 2015. URL <http://arxiv.org/abs/1506.02640>.

Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, y Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. En *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 779–788, 2016. ISBN 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.91.

Shaoqing Ren, Kaiming He, Ross B. Girshick, y Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *arXiv e-prints*, arXiv:1506.01497v2, 2016. URL <http://arxiv.org/abs/1506.01497>.

Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, y John Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. En *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94, pages 175–186, 1994. ISBN 0-89791-689-1. DOI: 10.1145/192844.192905.

Elaine Rich. *Artificial Intelligence*. McGraw-Hill, 1983. ISBN 0070522618.

Peter Richtárik y Martin Takáč. Parallel coordinate descent methods for big data optimization. *arXiv e-prints*, arXiv:1212.0873, 2012. URL <http://arxiv.org/abs/1212.0873>.

Peter Richtárik y Martin Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 156(1):433–484, Mar 2016. ISSN 1436-4646. DOI: 10.1007/s10107-015-0901-6.

Mark O. Riedl. The lovelace 2.0 test of artificial creativity and intelligence. *arXiv e-prints*, arXiv:1410.6142, 2014. URL <http://arxiv.org/abs/1410.6142>.

Martin Riedmiller. Advanced Supervised Learning in Multi-layer Perceptrons: From Backpropagation to Adaptive Learning Algorithms. *Computer Standards & Interfaces*, 16(3):265 – 278, 1994. ISSN 0920-5489. DOI: 10.1016/0920-5489(94)90017-5.

Martin Riedmiller y Heinrich Braun. A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. En *IEEE International Conference on Neural Networks*, pages 586–591 vol.1, 1993. DOI: 10.1109/ICNN.1993.298623.

Fred Rieke, Davd Warland, Rob de Ruyter van Steveninck, y William Bialek. *Spikes: Exploring the Neural Code*. MIT Press, 1999. ISBN 0262181746.

Sebastian Risi y Kenneth O. Stanley. An Enhanced Hypercube-Based Encoding for Evolving the Placement, Density, and Connectivity of Neurons. *Artificial Life*, 18(4):331–363, 2012. DOI: 10.1162/ARTL_a_00071.

Herbert Robbins y Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951. ISSN 0003-4851. URL <http://www.jstor.org/stable/2236626>.

Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, 1996. ISBN 3540605053. URL <https://page.mi.fu-berlin.de/rojas/neural/>.

Paul Romer. The trouble with macroeconomics. *The American Economist*, Commons Memorial Lecture of the Omicron Delta Epsilon Society, 2016. URL <https://paulromer.net/wp-content/uploads/2016/09/WP-Trouble.pdf>.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, y Yoshua Bengio. FitNets: Hints for Thin Deep Nets. *ICLR'2015*, arXiv:1412.6550, 2015. URL <http://arxiv.org/abs/1412.6550>.

Frank Rosenblatt. The perceptron: A perceiving and recognizing automaton. *Cornell Aeronautical Laboratory, Buffalo, New York*, Report 85-60-1, 1957. URL <https://goo.gl/17Fprk>.

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. ISSN 1939-1471. DOI: 10.1037/h0042519. URL <http://psycnet.apa.org/journals/rev/65/6/386>.

Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962. URL <http://www.dtic.mil/docs/citations/AD0256582>.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv e-prints*, arXiv:1609.04747, 2016. URL <http://arxiv.org/abs/1609.04747>.

David E. Rumelhart, Geoffrey E. Hinton, y Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, October 1986a. DOI: 10.1038/323533a0.

David E. Rumelhart, James L. McClelland, y the PDP Research Group, editores. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition - Volume 1: Foundations*. MIT Press, 1986b. ISBN 0262181207.

David E. Rumelhart, James L. McClelland, y the PDP Research Group, editores. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition - Volume 2: Psychological and Biological Models*. MIT Press, 1986c. ISBN 0262132184.

David E. Rumelhart, Geoffrey E. Hinton, y Ronald J. Williams. Learning Representations by Back-propagating Errors. En James A. Anderson

y Edward Rosenfeld, editores, *Neurocomputing: Foundations of Research*, pages 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0262010976.

David E. Rumelhart, Richard Durbin, Richard Golden, y Yves Chauvin. Backpropagation: The basic theory. En Yves Chauvin y David E. Rumelhart, editores, *Backpropagation: Theory, Architectures, and Applications*, pages 1–34. Lawrence Erlbaum Associates Inc., 1995. ISBN 0805812598.

Thomas Philip Runarsson y Magnus Thor Jonsson. Evolution and design of distributed learning rules. En *2000 IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks*, pages 59–63, 2000. DOI: 10.1109/ECNN.2000.886220.

David Ruppert. Efficient estimators from a slowly convergent Robbins-Monro process. Technical Report 781, School of Operations Research and Industrial Engineering, Cornell University, 1988. URL <https://goo.gl/VdvSzr>.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, y Fei-Fei Li. ImageNet Large Scale Visual Recognition Challenge. *arXiv e-prints*, arXiv:1409.0575, 2014. URL <http://arxiv.org/abs/1409.0575>.

Sirpa Saarinen, Randall Bramley, y George Cybenko. Ill-conditioning in neural network training problems. *SIAM Journal on Scientific Computing*, 14(3):693–714, 1993. DOI: 10.1137/0914044.

Sara Sabour, Nicholas Frosst, y Geoffrey E. Hinton. Dynamic routing between capsules. *arXiv e-prints*, arXiv:1710.09829, 2017. URL <http://arxiv.org/abs/1710.09829>.

Tetsuya Sakurai, Akira Imakura, Yuto Inoue, y Yasunori Futamura. Alternating optimization method based on nonnegative matrix factorizations for deep neural networks. *arXiv*, arXiv:1605.04639, 2016. URL <http://arxiv.org/abs/1605.04639>.

Ruslan Salakhutdinov y Geoffrey Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. En Marina Meila y Xiaotong Shen, editores, *AISTATS'2007 Proceedings of the 11th International Conference on Artificial Intelligence and Statistics*, volume 2 of *Proceedings of Machine Learning Research*, pages 412–419, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR. URL <http://proceedings.mlr.press/v2/salakhutdinov07a.html>.

Ruslan Salakhutdinov y Geoffrey Hinton. Deep boltzmann machines. En David van Dyk y Max Welling, editores, *AISTATS'2009 Proceedings of the 12th International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 448–455, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR. URL <http://proceedings.mlr.press/v5/salakhutdinov09a.html>.

Tariq Samad. Back propagation with expected source values. *Neural Networks*, 4(5):615 – 618, 1991. ISSN 0893-6080. DOI: 10.1016/0893-6080(91)90015-W.

Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959. ISSN 0018-8646. DOI: 10.1147/rd.33.0210.

Terence D. Sanger. Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural Networks*, 2(6):459–473, 1989. ISSN 0893-6080. DOI: 10.1016/0893-6080(89)90044-0.

Seiya Satoh y Ryohei Nakano. Fast and stable learning utilizing singular regions of multilayer perceptron. *Neural Processing Letters*, 38(2):99–115, October 2013. ISSN 1370-4621. DOI: 10.1007/s11063-013-9283-z.

Andrew M. Saxe, Pang Wei Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, y Andrew Y. Ng. On random weights and unsupervised feature learning. En *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 1089–1096. Omnipress, 2011. ISBN 978-1-4503-0619-5.

Andrew M. Saxe, James L. McClelland, y Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *ICLR'2014*, arXiv:1312.6120, 2014. URL <http://arxiv.org/abs/1312.6120>.

Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, y Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007. ISSN 0036-8075. DOI: 10.1126/science.1144079. URL <http://science.sciencemag.org/content/317/5844/1518>.

Tom Schaul y Yann LeCun. Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients. *ICLR'2013*, arXiv:1301.3764, 2013. URL <http://arxiv.org/abs/1301.3764>.

Tom Schaul, Sixin Zhang, y Yann LeCun. No more pesky learning rates. En Sanjoy Dasgupta y David McAllester, editores, *Proceedings of the 30th International Conference on Machine Learning*, volume 28(3)

of *Proceedings of Machine Learning Research*, pages 343–351, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/schaul13.html>.

Tom Schaul, Ioannis Antonoglou, y David Silver. Unit tests for stochastic optimization. *ICLR'2014*, arXiv:1312.6055, 2014. URL <http://arxiv.org/abs/1312.6055>.

Dominik Scherer, Andreas Müller, y Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. En Konstantinos Diamantaras, Wlodek Duch, y Lazaros S. Iliadis, editores, *Artificial Neural Networks – ICANN 2010*, pages 92–101, 2010. ISBN 978-3-642-15825-4. DOI: 10.1007/978-3-642-15825-4_10.

Bernhard Scholkopf y Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001. ISBN 0262194759.

Nicol N. Schraudolph, Jin Yu, y Simon Günter. A Stochastic Quasi-Newton Method for Online Convex Optimization. En Marina Meila y Xiaotong Shen, editores, *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, volume 2 de *Proceedings of Machine Learning Research*, pages 436–443, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR. URL <http://proceedings.mlr.press/v2/schraudolph07a.html>.

Florian Schroff, Dmitry Kalenichenko, y James Philbin. Facenet: A unified embedding for face recognition and clustering. En *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 815–823, 2015. ISBN 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298682.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, y Philipp Moritz. Trust region policy optimization. En *Proceedings of the 32nd International Conference on Machine Learning*, pages 1889–1897. PMLR, 2015a. URL <http://proceedings.mlr.press/v37/schulman15.html>.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, y Pieter Abbeel. Trust Region Policy Optimization. *arXiv*, arXiv:1502.05477, 2015b. URL <http://arxiv.org/abs/1502.05477>.

Michael Schuster. *On Supervised Learning from Sequential Data with Applications for Speech Recognition*. PhD thesis, Nara Institute of Science and Technology, 1999.

Pedro Schwartz. The fear of robots. *Econlib: Library of Economics and Liberty*, May 1, 2017. URL <http://www.econlib.org/library/Columns/y2017/Schwartzrobots.html?>

Holger Schwenk y Yoshua Bengio. Training methods for adaptive boosting of neural networks. En M. I. Jordan, M. J. Kearns, y S. A. Solla, editores, *NIPS'1997 Advances in Neural Information Processing Systems 10*, pages 647–653. MIT Press, 1997. URL <https://goo.gl/2gvkzg>.

Terrence J. Sejnowski y Charles R. Rosenberg. Parallel Networks that Learn to Pronounce English Text. *Complex Systems*, 1(1):145–168, 1987. ISSN 0891-2513. URL http://www.complex-systems.com/abstracts/v01_i01_a10.html.

Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, y Yann LeCun. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. *arXiv e-prints*, arXiv:1312.6229, 2013. URL <http://arxiv.org/abs/1312.6229>.

Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948. ISSN 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

Evan Shelhamer, Jonathan Long, y Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):640–651, 2017. ISSN 0162-8828. DOI: 10.1109/TPAMI.2016.2572683.

Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, y Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. *arXiv e-prints*, arXiv:1609.05158, 2016. URL <http://arxiv.org/abs/1609.05158>.

Avanti Shrikumar, Peyton Greenside, y Anshul Kundaje. Learning important features through propagating activation differences. *arXiv e-prints*, arXiv:1704.02685, 2017. URL <http://arxiv.org/abs/1704.02685>.

Jocelyn Sietsma y Robert J. F. Dow. Neural Net Pruning - Why and How. En *IEEE 1988 International Conference on Neural Networks*, volume 1, pages 325–333, July 1988. DOI: 10.1109/ICNN.1988.23864.

Jocelyn Sietsma y Robert J.F. Dow. Creating artificial neural networks that generalize. *Neural Networks*, 4(1):67–79, 1991. ISSN 0893-6080. DOI: 10.1016/0893-6080(91)90033-2.

Joseph Sill, Gábor Takács, Lester W. Mackey, y David Lin. Feature-weighted linear stacking. *arXiv e-prints*, 2009. URL <http://arxiv.org/abs/0911.0460>.

Fernando M. Silva y Luis B. Almeida. Speeding up Backpropagation. En R. Eckmiller, editor, *Advanced Neural Computers - Proceedings of the*

International Symposium on Neural Networks for Sensory and Motor Systems, page 151–158. North-Holland, 1990. ISBN 978-0-444-88400-8. DOI: 10.1016/B978-0-444-88400-8.50022-4.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, y Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. DOI: 10.1038/nature16961.

Jiri Sima. Back-propagation is not efficient. *Neural Networks*, 9(6):1017–1023, 1996. ISSN 0893-6080. DOI: 10.1016/0893-6080(95)00135-2.

Jiri Sima. Training a single sigmoidal neuron is hard. *Neural Computation*, 14(11):2709–2728, Nov 2002. ISSN 0899-7667. DOI: 10.1162/089976602760408035.

Patrice Simard, Bernard Victorri, Yann LeCun, y John Denker. Tangent Prop - A formalism for specifying selected invariances in an adaptive network. En J. E. Moody, S. J. Hanson, y R. P. Lippmann, editores, *NIPS'2001 Advances in Neural Information Processing Systems 4*, pages 895–903. Morgan-Kaufmann, 1992. URL <https://goo.gl/N3hQhR>.

Patrice Y. Simard, Dave Steinkraus, y John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. En *ICDAR'2003 Proceedings of the 7th International Conference on Document Analysis and Recognition*, pages 958–963, Aug 2003. DOI: 10.1109/ICDAR.2003.1227801.

Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, 3rd edition, 1996. ISBN 0262691914.

Karen Simonyan y Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR'2015*, arXiv:1409.1556, 2014. URL <http://arxiv.org/abs/1409.1556>.

Yoram Singer y John C. Duchi. Efficient Learning using Forward-Backward Splitting. En Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, y A. Culotta, editores, *NIPS'2009 Advances in Neural Information Processing Systems 22*, pages 495–503. Curran Associates, Inc., 2009. URL <https://goo.gl/cnhvT4>.

J. Sjöberg y Lennart Ljung. Overtraining, regularization and searching for a minimum, with application to neural networks. *International Journal of Control*, 62(6):1391–1407, 1995. DOI: 10.1080/00207179508921605.

David M. Skapura. *Building Neural Networks*. ACM Press / Addison-Wesley Professional, 1993. ISBN 0201539217.

Jasper Snoek, Hugo Larochelle, y Ryan P Adams. Practical bayesian optimization of machine learning algorithms. En F. Pereira, C. J. C. Burges, L. Bottou, y K. Q. Weinberger, editores, *NIPS'2012 Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012. URL <https://goo.gl/dZsHjf>.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat, y Ryan P. Adams. Scalable bayesian optimization using deep neural networks. En *ICML'2015: Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 6-11 July 2015*, pages 2171–2180, 2015. URL <http://jmlr.org/proceedings/papers/v37/snoek15.html>.

Richard Socher, Milind Ganjoo, Christopher D. Manning, y Andrew Y. Ng. Zero-shot learning through cross-modal transfer. En *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, pages 935–943, USA, 2013. Curran Associates Inc. URL <https://arxiv.org/abs/1301.3666>.

Jascha Sohl-Dickstein, Ben Poole, y Surya Ganguli. An adaptive low dimensional quasi-Newton sum of functions optimizer. *arXiv e-prints*, arXiv:1311.2115, 2013. URL <http://arxiv.org/abs/1311.2115>.

Jascha Sohl-Dickstein, Ben Poole, y Surya Ganguli. Fast large-scale optimization by unifying stochastic gradient and quasi-Newton methods. En Eric P. Xing y Tony Jebara, editores, *Proceedings of the 31st International Conference on Machine Learning*, volume 32(2) of *Proceedings of Machine Learning Research*, pages 604–612, Bejing, China, 22–24 Jun 2014. PMLR. URL <http://proceedings.mlr.press/v32/sohl-dicksteinb14.html>.

Sara A. Solla, Esther Levin, y Michael Fleisher. Parallel Networks that Learn to Pronounce English Text. *Complex Systems*, 2(6):625–639, 1988. ISSN 0891-2513. URL <http://www.complex-systems.com/pdf/02-6-1.pdf>.

Eduardo D. Sontag. Feedback stabilization using two-hidden-layer nets. Technical Report SYCON-90-11, Rutgers Center for Systems and Control, 1990.

Eduardo D. Sontag. Feedback stabilization using two-hidden-layer nets. *IEEE Transactions on Neural Networks*, 3(6):981–990, Nov 1992. ISSN 1045-9227. DOI: 10.1109/72.165599.

Danny C. Sorensen. Newton's method with a model trust region modification. *SIAM Journal on Numerical Analysis*, 19(2):409–426, 1982. DOI: 10.1137/0719026.

Alessandro Sperduti y Antonina Starita. Speed up learning and network optimization with extended back propagation. *Neural Networks*, 6(3):365–383, 1993. ISSN 0893-6080. DOI: 10.1016/0893-6080(93)90004-G.

Olaf Sporns. *Discovering the Human Connectome*. MIT Press, 2012. ISBN 0262017903.

Olaf Sporns. *Networks of the Brain*. MIT Press, 2013. ISBN 0262017903.

Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, y Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv e-prints*, arXiv:1412.6806, 2014. URL <http://arxiv.org/abs/1412.6806>.

Nathan Srebro y Adi Shraibman. Rank, Trace-Norm and Max-Norm. En Peter Auer y Ron Meir, editores, *COLT 2005: Proceedings of the 18th Annual Conference on Learning Theory, COLT 2005, Bertinoro, Italy, June 27-30, 2005.*, pages 545–560, 2005. ISBN 978-3-540-31892-7. DOI: 10.1007/11503415_37.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, y Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL <http://jmlr.org/papers/v15/srivastava14a.html>.

Rupesh Kumar Srivastava, Klaus Greff, y Jürgen Schmidhuber. Training very deep networks. *arXiv e-prints*, abs/1507.06228, 2015a. URL <http://arxiv.org/abs/1507.06228>.

Rupesh Kumar Srivastava, Klaus Greff, y Jürgen Schmidhuber. Highway networks. *ICML'2015 Deep Learning Workshop*, arXiv:1505.00387, 2015b. URL <http://arxiv.org/abs/1505.00387>.

Rupesh Kumar Srivastava, Klaus Greff, y Jürgen Schmidhuber. Training very deep networks. En Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, y Roman Garnett, editores, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2377–2385, 2015c. URL <http://papers.nips.cc/paper/5850-training-very-deep-networks>.

Johannes Stallkamp, Marc Schlipsing, Jan Salmen, y Christian Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. En *IEEE International Joint Conference on Neural Networks*, pages 1453–1460, 2011. DOI: 10.1109/IJCNN.2011.6033395.

Kenneth O. Stanley y Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, June 2002. ISSN 1063-6560. DOI: 10.1162/106365602320169811.

Kenneth O. Stanley, David B. D'Ambrosio, y Jason Gauci. A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial Life*, 15(2):185–212, 2009. DOI: 10.1162/artl.2009.15.2.15202.

Peter Sterling y Simon Laughlin. *Principles of Neural Design*. MIT Press, 2015. ISBN 0262028700.

Jiawei Su, Danilo Vasconcellos Vargas, y Kouichi Sakurai. One pixel attack for fooling deep neural networks. *arXiv e-prints*, arXiv:1710.08864, 2017. URL <http://arxiv.org/abs/1710.08864>.

Yusuke Sugomori. *Java Deep Learning Essentials*. Packt Publishing, 1st edition, 2016. ISBN 1785282190.

Sainbayar Sukhbaatar, arthur szlam, Jason Weston, y Rob Fergus. End-to-end memory networks. En C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, y R. Garnett, editores, *Advances in Neural Information Processing Systems 28*, pages 2440–2448. Curran Associates, Inc., 2015. URL <https://goo.gl/PQRALU>.

David Sussillo. Random walk initialization for training very deep feedforward networks. *arXiv e-prints*, arXiv:1412.6558, 2014. URL <http://arxiv.org/abs/1412.6558>.

Ilya Sutskever, James Martens, George Dahl, y Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. En Sanjoy Dasgupta y David McAllester, editores, *ICML'13 Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/sutskever13.pdf>.

Richard S. Sutton y Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1st edition, 1998. ISBN 0262193981.

Kevin Swersky, Jasper Snoek, y Ryan Prescott Adams. Freeze-thaw bayesian optimization. *arXiv e-prints*, arXiv:1406.3896, 2014. URL <http://arxiv.org/abs/1406.3896>.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, y Andrew Rabinovich. Going deeper with convolutions. *arXiv e-prints*, arXiv:1409.4842, 2014a. URL <http://arxiv.org/abs/1409.4842>.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, y Rob Fergus. Intriguing properties of neural networks. *ICLR'2014*, arXiv:1312.6199, 2014b. URL <http://arxiv.org/abs/1312.6199>.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, y Andrew Rabinovich. Going deeper with convolutions. En *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9, 2015a. ISBN 978-1-4673-6964-0. doi: 10.1109/CVPR.2015.7298594.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, y Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *arXiv e-prints*, arXiv:1512.00567, 2015b. URL <http://arxiv.org/abs/1512.00567>.

Christian Szegedy, Sergey Ioffe, y Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv e-prints*, arXiv:1602.07261, 2016. URL <http://arxiv.org/abs/1602.07261>.

Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, y Lior Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. En *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 1701–1708, June 2014. ISBN 978-1-4799-5118-5. doi: 10.1109/CVPR.2014.220.

Gábor Takács, István Pilászy, Bottyán Németh, y Domonkos Tikk. Matrix Factorization and Neighbor Based Algorithms for the Netflix Prize Problem. En *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pages 267–274, 2008. ISBN 978-1-60558-093-7. doi: 10.1145/1454008.1454049.

Pang-Ning Tan, Michael Steinbach, y Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 1st edition, 2005. ISBN 0321321367.

Yichuan Tang y Chris Eliasmith. Deep networks for robust visual recognition. En *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 1055–1062, USA, 2010. Omnipress. ISBN 978-1-60558-907-7. URL <http://icml2010.haifa.il.ibm.com/papers/370.pdf>.

Joshua B. Tenenbaum, Vin de Silva, y John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000. ISSN 0036-8075. doi: 10.1126/science.290.5500.2319.

Gerald Tesauro y Bob Janssens. Scaling Relationships in Back-propagation Learning. *Complex Systems*, 2(1):39—44, 1988. ISSN 0891-2513. URL http://www.complex-systems.com/abstracts/v02_i01_a03.html.

G. Thimm y E. Fiesler. High-order and multilayer perceptron initialization. *IEEE Transactions on Neural Networks*, 8(2):349–359, Mar 1997. ISSN 1045-9227. DOI: 10.1109/72.557673.

Robert Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society (Series B: Methodological)*, 58(1):267–288, 1996. URL <http://www.jstor.org/stable/2346178>.

Robert Tibshirani. Regression Shrinkage and Selection via the Lasso: A retrospective. *Journal of the Royal Statistical Society (Series B: Statistical Methodology)*, 73(3):273–282, 2011. ISSN 1467-9868. DOI: 10.1111/j.1467-9868.2011.00771.x.

Andrey Nikolaevich Tikhonov y Vasiliy Yakovlevich Arsenin. *Solutions of Ill-Posed Problems*. Scripta Series in Mathematics. V.H. Winston & Sons, 1977. ISBN 0470991240.

Tom Tollenaere. SuperSAB: Fast adaptive back propagation with good scaling properties. *Neural Networks*, 3(5):561 – 573, 1990. ISSN 0893-6080. DOI: 10.1016/0893-6080(90)90006-7.

Nicholas K. Treadgold y Tamas D. Gedeon. Simulated annealing and weight decay in adaptive learning: the SARPROP algorithm. *IEEE Transactions on Neural Networks*, 9(4):662–668, Jul 1998. ISSN 1045-9227. DOI: 10.1109/72.701179.

Bipin Kumar Tripathi y Prem Kumar Kalra. On efficient learning machine with root-power mean neuron in complex domain. *IEEE Transactions on Neural Networks*, 22(5):727–738, May 2011. ISSN 1045-9227. DOI: 10.1109/TNN.2011.2115251.

Edward R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, Connecticut, 1990. ISBN 0961392118.

Edward R. Tufte. *Visual Explanations*. Graphics Press, Cheshire, Connecticut, 1997. ISBN 0961392126.

Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, 2001. ISBN 0961392142.

Edward R. Tufte. *Beautiful Evidence*. Graphics Press, LLC, first edition, May 2006. ISBN 0961392177.

Joseph Turian, Lev Ratinov, y Yoshua Bengio. Word Representations: A Simple and General Method for Semi-supervised Learning. En *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 384–394, 2010. URL <http://www.aclweb.org/anthology/P10-1040>.

Alan M. Turing. Intelligent Machinery. Technical report, National Physical Laboratory, London, 1948. URL http://www.alanturing.net/intelligent_machinery/.

Alan M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950. ISSN 00264423. DOI: 10.1093/mind/LIX.236.433. URL <http://www.jstor.org/stable/2251299>.

Paul E. Utgoff. Perceptron trees: A case study in hybrid concept representations. En Howard E. Shrobe, Tom M. Mitchell, y Reid G. Smith, editores, *Proceedings of the 7th National Conference on Artificial Intelligence. St. Paul, MN, August 21-26, 1988.*, pages 601–606. AAAI Press / The MIT Press, 1988. ISBN 0-262-51055-3. URL <http://www.aaai.org/Papers/AAAI/1988/AAAI88-107.pdf>.

Paul E. Utgoff. Perceptron trees: A case study in hybrid concept representations. *Connection Science*, 1(4):377–391, 1989. DOI: 10.1080/09540098908915648.

Harri Valpola. From neural {PCA} to deep unsupervised learning. En Ella Bingham, Samuel Kaski, Jorma Laaksonen, y Jouko Lampinen, editores, *Advances in Independent Component Analysis and Learning Machines, Chapter 8*, pages 143 – 171. Academic Press, 2015. ISBN 978-0-12-802806-3. DOI: 10.1016/B978-0-12-802806-3.00008-7.

Laurens van der Maaten y Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. URL <http://www.jmlr.org/papers/v9/vandermaaten08a.html>.

P. Patrick van der Smagt. Minimisation methods for training feed-forward neural networks. *Neural Networks*, 7(1):1 – 11, 1994. ISSN 0893-6080. DOI: 10.1016/0893-6080(94)90052-3.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, y Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. En *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, 2008. ISBN 978-1-60558-205-4. DOI: 10.1145/1390156.1390294.

Oriol Vinyals, Alexander Toshev, Samy Bengio, y Dumitru Erhan. Show and Tell: A Neural Image Caption Generator. *arXiv e-prints*, arXiv:1411.4555, 2014. URL <http://arxiv.org/abs/1411.4555>.

Oriol Vinyals, Charles Blundell, Timothy P. Lillicrap, Koray Kavukcuoglu, y Daan Wierstra. Matching Networks for One Shot Learning. *arXiv e-prints*, arXiv:1606.04080, 2016. URL <http://arxiv.org/abs/1606.04080>.

Paul Viola y Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004. ISSN 1573-1405. DOI: 10.1023/B:VISI.0000013087.49260.fb.

Javier E. Vitela y Jaques Reifman. Premature saturation in backpropagation networks: Mechanism and necessary conditions. *Neural Networks*, 10(4):721–735, June 1997. ISSN 0893-6080. DOI: 10.1016/S0893-6080(96)00117-7.

T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, y D. L. Alkon. Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, 59(4):257–263, Sep 1988. ISSN 1432-0770. DOI: 10.1007/BF00332914.

Ulrike von Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17(4):395–416, 2007. DOI: 10.1007/s11222-007-9033-z.

John von Neumann y Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 60th anniversary commemorative edition edition, 1947. ISBN 0691130612.

Stefan Wager, Sida Wang, y Percy S. Liang. Dropout training as adaptive regularization. En C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, y K. Q. Weinberger, editores, *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS’13, pages 351–359. Curran Associates Inc., 2013. URL <https://goo.gl/fmekb8>.

Eric Wan. Time Series Prediction by Using a Connectionist Network with Internal Delay Lines. En A. Weigend y N. Gershenfeld, editores, *Time Series Prediction: Forecasting the Future and Understanding the Past. SFI Studies in the Sciences of Complexity*, pages 195–217. Addison-Wesley, 1994.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, y Rob Fergus. Regularization of Neural Networks using DropConnect. En Sanjoy Dasgupta y David McAllester, editores, *Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) of *Proceedings of Machine Learning Research*, pages 1058–1066, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/wan13.html>.

Jian Wang, Jie Yang, y Wei Wu. Convergence of cyclic and almost-cyclic learning with momentum for feedforward neural networks. *IEEE Transactions on Neural Networks*, 22(8):1297–1306, Aug 2011. ISSN 1045-9227. DOI: 10.1109/TNN.2011.2159992.

Sida Wang y Christopher Manning. Fast dropout training. En Sanjoy Dasgupta y David McAllester, editores, *Proceedings of the 30th International Conference on Machine Learning*, volume 28(2) de *Proceedings of Machine Learning Research*, páginas 118–126, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL <http://proceedings.mlr.press/v28/wang13a.html>.

X.G. Wang, Z. Tang, H. Tamura, y M. Ishii. A modified error function for the backpropagation algorithm. *Neurocomputing*, 57:477 – 484, 2004. ISSN 0925-2312. DOI: 10.1016/j.neucom.2003.12.006. New Aspects in Neurocomputing: 10th European Symposium on Artificial Neural Networks 2002.

David Warde-Farley, Ian J. Goodfellow, Aaron Courville, y Yoshua Bengio. An empirical analysis of dropout in piecewise linear networks. En *ICLR'2014 International Conference on Learning Representations*, volumen arXiv:1312.6197, 2014. URL <http://arxiv.org/abs/1312.6197>.

Philip D. Wasserman. *Neural Computing: Theory and Practice*. Van Nostrand Reinhold, 1989. ISBN 0442207433.

Philip D. Wasserman. *Advanced Methods in Neural Computing*. Van Nostrand Reinhold, 1993. ISBN 0442004613.

Andreas S. Weigend, David E. Rumelhart, y Bernardo A. Huberman. Generalization by weight-elimination with application to forecasting. En *NIPS'1990 Advances in Neural Information Processing Systems 3*, páginas 875–882. Morgan-Kaufmann, 1990. URL <https://goo.gl/864Mto>.

Andreas Wendemuth. Learning the unlearnable. *Journal of Physics A: Mathematical and General*, 28(18):5423, 1995. URL <http://stacks.iop.org/0305-4470/28/i=18/a=030>.

Paul John Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974. URL <https://www.researchgate.net/publication/35657389>.

Paul John Werbos. Applications of advances in nonlinear sensitivity analysis. En R. F. Drenick y F. Kozin, editores, *System Modeling and Optimization: Proceedings of the 10th IFIP Conference New York City, USA, August 31 – September 4, 1981*, páginas 762–770. Springer, 1982. ISBN 978-3-540-39459-4. DOI: 10.1007/BFb0006203.

Paul John Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. John Wiley & Sons, 1994. ISBN 0471598976.

Lodewyk F. A. Wessels y Etienne Barnard. Avoiding false local minima by proper initialization of connections. *IEEE Transactions on Neural Networks*, 3(6):899–905, Nov 1992. ISSN 1045-9227. DOI: 10.1109/72.165592.

Bernard Widrow y Marcian E. Hoff. Adaptive switching circuits. En *1960 IRE WESCON Convention Record, Part 4*, pages 96–104, New York, August 1960. Institute of Radio Engineers, Institute of Radio Engineers. URL <http://www-isl.stanford.edu/~widrow/papers/c1960adaptiveswitching.pdf>.

Bernard Widrow y Michael A. Lehr. 30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, Sep 1990. ISSN 0018-9219. DOI: 10.1109/5.58323.

Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, y Benjamin Recht. The Marginal Value of Adaptive Gradient Methods in Machine Learning. *arXiv e-prints*, arXiv:1705.08292, 2017. URL <http://arxiv.org/abs/1705.08292>.

D.Randall Wilson y Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429 – 1451, 2003. ISSN 0893-6080. DOI: 10.1016/S0893-6080(03)00138-2.

Robert O. Winder. Threshold logic asymptotes. *IEEE Transactions on Computers*, C-19(4):349–353, April 1970. ISSN 0018-9340. DOI: 10.1109/T-C.1970.222921.

Rodney Winter y Bernard Widrow. MADALINE RULE II: A training algorithm for neural networks. En *IEEE International Conference on Neural Networks*, volume 1, pages 401–408, July 1988. DOI: 10.1109/ICNN.1988.23872. URL <http://www-isl.stanford.edu/people/widrow/papers/c1988madalinerule.pdf>.

David H. Wolpert. Stacked generalization. *Neural Networks*, 5(2): 241–259, February 1992. ISSN 0893-6080. DOI: 10.1016/S0893-6080(05)80023-1.

David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996. ISSN 0899-7667. DOI: 10.1162/neco.1996.8.7.1341.

David H. Wolpert y William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, Apr 1997. ISSN 1089-778X. DOI: 10.1109/4235.585893.

Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, y Gang Sun. Deep image: Scaling up image recognition. *arXiv e-prints*, arXiv:1501.02876, 2015. URL <http://arxiv.org/abs/1501.02876>. Withdrawn.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, y Jeffrey Dean. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv e-prints*, 2016. URL <http://arxiv.org/abs/1609.08144>.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, y Yoshua Bengio. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *CoRR*, abs/1502.03044, 2015. URL <http://arxiv.org/abs/1502.03044>.

Zong-Ben Xu, Rui Zhang, y Wen-Feng Jing. When Does Online BP Training Converge? *IEEE Transactions on Neural Networks*, 20(10):1529–1539, Oct 2009. ISSN 1045-9227. DOI: 10.1109/TNN.2009.2025946.

Paul Viola y Michael J. Jones. Robust real-time object detection. Technical Report CLR 2001/01, Cambridge Research Laboratory, Compaq, February 2001. URL <http://www.hpl.hp.com/techreports/Compaq-DEC/CRL-2001-1.pdf>.

J. Y. F. Yam y T. W. S. Chow. Feedforward networks training speed enhancement by optimal initialization of the synaptic coefficients. *IEEE Transactions on Neural Networks*, 12(2):430–434, Mar 2001. ISSN 1045-9227. DOI: 10.1109/72.914538.

Liping Yang y Wanzen Yu. Backpropagation with homotopy. *Neural Computation*, 5(3):363–366, May 1993. ISSN 0899-7667. DOI: 10.1162/neco.1993.5.3.363.

Jason Yosinski, Jeff Clune, Yoshua Bengio, y Hod Lipson. How transferable are features in deep neural networks? En *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS’14, pages 3320–3328. MIT Press, 2014. URL <https://goo.gl/AZacCB>.

W. J. Youden. Index for rating diagnostic tests. *Cancer*, 3(1):32–35, 1950. ISSN 1097-0142. DOI: 10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3.

Dong Yu y Li Deng. Deep convex net: A scalable architecture for speech pattern classification. En *INTERSPEECH 2011, 12th Annual Conference of the International Speech Communication Association, Florence, Italy, August 27-31, 2011*, pages 2285–2288. ISCA, 2011. URL <https://goo.gl/cbvEi9>.

Dong Yu, Shizhen Wang, y Li Deng. Sequential labeling using deep-structured conditional random fields. *IEEE Journal of Selected Topics in Signal Processing*, 4(6):965–973, Dec 2010. ISSN 1932-4553. DOI: 10.1109/JSTSP.2010.2075990.

Fisher Yu y Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *ICLR’2016*, arXiv:1511.07122, 2015. URL <http://arxiv.org/abs/1511.07122>.

Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, y Inderjit Dhillon. Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems. En *Proceedings of the 2012 IEEE 12th International Conference on Data Mining, ICDM’2012*, pages 765–774, 2012. ISBN 978-0-7695-4905-7. DOI: 10.1109/ICDM.2012.168.

Kai Yu, Shenghuo Zhu, John Lafferty, y Yihong Gong. Fast Nonparametric Matrix Factorization for Large-scale Collaborative Filtering. En *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR’2009*, pages 211–218, 2009. ISBN 978-1-60558-483-6. DOI: 10.1145/1571941.1571979.

Xiao-Hu Yu y Guo-An Chen. Efficient backpropagation learning using optimal learning rate and momentum. *Neural Networks*, 10(3):517 – 527, 1997. ISSN 0893-6080. DOI: 10.1016/S0893-6080(96)00102-5.

Xiao-Hu Yu, Guo-An Chen, y Shi-Xin Cheng. Dynamic learning rate optimization of the backpropagation algorithm. *IEEE Transactions on Neural Networks*, 6(3):669–677, May 1995. ISSN 1045-9227. DOI: 10.1109/72.377972.

Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv e-prints*, arXiv:1212.5701, 2012. URL <http://arxiv.org/abs/1212.5701>.

Matthew D. Zeiler y Rob Fergus. Visualizing and understanding convolutional networks. *arXiv e-prints*, arXiv:1311.2901, 2013. URL <http://arxiv.org/abs/1311.2901>.

ChengXiang Zhai y Sean Massung. *Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining.* ACM Books, Morgan & Claypool, 2016. ISBN 1970001194.

Richard Zhang, Phillip Isola, y Alexei A. Efros. Colorful image colorization. *arXiv e-prints*, arXiv:1603.08511, 2016. URL <http://arxiv.org/abs/1603.08511>.

Richard Zhang, Jun-Yan Zhu, Phillip Isola, Xinyang Geng, Angela S. Lin, Tianhe Yu, y Alexei A. Efros. Real-time user-guided image colorization with learned deep priors. *arXiv e-prints*, arXiv:1705.02999, 2017a. URL <http://arxiv.org/abs/1705.02999>.

Richard Zhang, Jun-Yan Zhu, Phillip Isola, Xinyang Geng, Angela S. Lin, Tianhe Yu, y Alexei A. Efros. Real-time user-guided image colorization with learned deep priors. *ACM Transactions on Graphics*, 36(4):119:1–119:11, 2017b. DOI: 10.1145/3072959.3073703.

Rui Zhang, Zong-Ben Xu, Guang-Bin Huang, y Dianhui Wang. Global Convergence of Online BP Training with Dynamic Learning Rate. *IEEE Transactions on Neural Networks and Learning Systems*, 23(2):330–341, Feb 2012. ISSN 2162-237X. DOI: 10.1109/TNNLS.2011.2178315.

Sixin Zhang, Anna E Choromanska, y Yann LeCun. Deep learning with Elastic Averaging SGD. En C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, y R. Garnett, editores, *NIPS’2015 Advances in Neural Information Processing Systems 28*, pages 685–693. Curran Associates, Inc., 2015. URL <https://goo.gl/z1REBz>.

Tianhao Zhang, Zoe McCarthy, Owen Jow, Dennis Lee, Ken Goldberg, y Pieter Abbeel. Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. *arXiv e-prints*, arXiv:1710.04615, 2017c. URL <http://arxiv.org/abs/1710.04615>.

Alice X. Zheng y Mikhail Bilenko. Lazy paired hyper-parameter tuning. En Francesca Rossi, editor, *IJCAI’2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 1924–1931. IJCAI/AAAI, January 2013. ISBN 978-1-57735-633-2. URL <https://www.ijcai.org/Abstract/13/284>.

Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, y Aude Oliva. Learning deep features for scene recognition using places database. En *NIPS’2014 Advances in Neural Information Processing Systems 27*, pages 487–495, 2014. URL <https://goo.gl/wkSGRu>.

Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, y Antonio Torralba. Places: A 10 million Image Database for Scene Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP(99):1–1, 2017. ISSN 0162-8828. DOI: 10.1109/TPAMI.2017.2723009.

Yi-Tong Zhou y Rama Chellappa. Computation of optical flow using a neural network. En *IEEE 1988 International Conference on Neural Networks*, volume 2, pages 71–78, 1988. DOI: 10.1109/ICNN.1988.23914.

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, y Rong Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. En *AAIM'2008 Algorithmic Aspects in Information and Management*, pages 337–348, 2008. ISBN 978-3-540-68880-8. DOI: 10.1007/978-3-540-68880-8_32.

Jun-Yan Zhu, Philipp Krähenbühl, Eli Shechtman, y Alexei A. Efros. Generative visual manipulation on the natural image manifold. *arXiv e-prints*, arXiv:1609.03552, 2016. URL <http://arxiv.org/abs/1609.03552>.

Jun-Yan Zhu, Taesung Park, Phillip Isola, y Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv e-prints*, arXiv:1703.10593, 2017. URL <http://arxiv.org/abs/1703.10593>.

Hui Zou y Trevor Hastie. Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society (Series B: Statistical Methodology)*, 67(2):301–320, 2005. DOI: 10.1111/j.1467-9868.2005.00503.x.

Y.H. Zweiri, J.F. Whidborne, y L.D. Seneviratne. A three-term back-propagation algorithm. *Neurocomputing*, 50:305 – 318, 2003. ISSN 0925-2312. DOI: 10.1016/S0925-2312(02)00569-6.

Índice alfabético

- óptimo local, 483
- 0-1 loss, 62, 473
- 0.632-bootstrap, 86
- 1 vs. all, 61, 243
- 3 en raya, 13

- A*, 12
- Abbeel, Pieter, 9
- abducción, 103
- ABP, Adaptive BackPropagation, 397
- abstracción, 146
 - niveles, 149
- accuracy, 61
- AdaBoost, 133
- AdaDelta, 501
- AdaGrad, 500
 - AdaGrad-RDA, 514
- ADALINE, 238
- Adam, 511
 - AdaMax, 513
 - NAdam, 513
- Adam (sistema), 545
- AdaMax, 513
- adaptación del dominio, 391
- AdaptiveRevision, 546
- AdaSecant, 509
- AdaTron, 242
- adicción, 166
- ADMM, 550
- ADN, 119
- adversarial examples, 372
- adversarial training, 373
- afilado de imágenes, 612
- ajedrez, 14, 123
- alcohol, 166
- alelo, 119
- AlexNet, 145, 659
- algoritmo
 - A*, 12, 20
 - advenedizo, 321
 - Apriori, 51
 - búsqueda de Grover, 23
 - de la torre, 321
 - del mosaico, 321
 - EM, 112
- algoritmo evolutivo
 - baldwiniano, 120
 - lamarckiano, 119
- algoritmo genético, 120
- algoritmos
 - basados en colonias de hormigas, 24
 - genéticos, 24
 - GRASP, 24
 - meméticos, 24
- ALife, 426
- Allanson, Jack T., 221
- ALMA, 248
- AlphaGo, 58
- ALS, 549
- Altavista, 38
- Amazon, 33, 69
 - Alexa, 40
 - Mechanical Turk, 43
 - sistema de recomendación, 107
- AMD, 152, 314
- ampliación del conjunto de datos, 367
- análisis, 325
 - ablativo, 326
 - error, 326
- análisis de componentes principales, 138
- análisis discriminante, 108
- análisis discriminante cuadrático, 108
- análisis discriminante lineal, 108
- análisis exploratorio de datos, 42
- analogía, 103
- anclaje, 130
- ángulo

- de paralelismo, 201
- trascendental, 200
- Anlauf, J.K., 242
- API
 - CUDA, 313
 - DirectCompute, 313
 - OpenCL, 313
- Apple
 - Siri, 40
- aprendizaje
 - curricular, 375
 - de representaciones, 166
 - en un intento, 392
 - multitarea, 166, 184, 391
 - online, 302
 - por asociación, 54, 165
 - por imitación, 56
 - por lotes, 303
 - por refuerzo, 52, 165
 - por transferencia, 388, 391, 645
 - sin datos, 392
 - social, 56
 - vicario, 56
- aprendizaje automático, 27
 - aplicaciones, 30
 - evaluación, 59
 - simbólico, 97
 - técnicas, 95
 - tipos, 34
- aprendizaje de representaciones, 341
- aprendizaje hebbiano, 163, 319
- aprendizaje no supervisado, 42
- aprendizaje por analogía, 34
- aprendizaje por imitación, 34
- aprendizaje supervisado, 35
- Apriori
 - algoritmo, 51
 - propiedad, 50
- aproximación estocástica, 395

- aproximador universal, 127, 149, 327, 329
perceptrón, 235
teorema, 327
Aranzio, Giulio Cesare, 168
árboles de decisión, 98
inducción, 99
poda, 100
regla de división, 99
area bajo la curva
área bajo la curva, AUC, 70
Ariely, Dan, 130
Aristóteles, 30
Arrow, Kenneth, 48
ART, Adaptive Resonance Theory, 206, 604
ART, Association Rule Tree, 51, 102
arte, 679
Ashby, Ross, 221
Asimov, Isaac, 6
asociación, 49
ASR, Automatic Speech Recognition, 452
Atari, 58
atención, 594
ATI, 152
ATP, adenosín trifosfato, 563
AUC, 70
autoencoder
eliminación de ruido, 453, 454
AVX, 152
axón, 185
- Bach, Johann Sebastian, 681
backgammon, 15
background removal, 676
backpropagation, 127, 223, 259
algoritmo, 285
gradiente, 291
historia, 315
implementación, 294
momento, 301
receta, 290
valor esperado, 300
backpropagation doble, 375
Bacon, Francis, 30
bagging, 131
Baidu
Minwa, 546
ban, 560
Bastiat, Frédéric, 7
bastones, 173
batch normalization, 409, 455, 643
Baumol, William, 3
bayesiano, 156
BCD, 549
BCI, Brain-Computer Interface, 186
bebés, 57
Bellman, Richard, 134
Bengio, Yoshua, 128, 144, 148, 324, 340, 350, 382, 510
Berkeley, George, 175
Bernoulli, Johann, 315
Bernstein, Sergei Natanovich, 327
betweenness, 47
Beurle, Raymond Louis, 221
Bézier, Pierre, 327
BFGS, 529, 530
BFGS sin memoria, 533
bias, 191, 192
bias, sesgo, 25
Biehl, Michael, 242
Bienert, Peter, 123
big data, 29
Bing, 38
bioinspiración, 160
BIRCH, 46
bit, 560
BKG9.8, 15
BLAS, 314
Blei, David, 140
blending, 134
Bliss, Timothy, 319
Block, Henry David, 237
Blondie24, 123
Blondie25, 123
Blum, Avrim, 331
BOAT, 101
bold-driver, 396
Boltzmann, Ludwig, 561
bonobos, 57
Bouch, Grady, 146
bookmaker informedness, 68
Boole, George, 128
boosting, 132
bootstrapping, 86, 131
Bordes, Antoine, 350
Box, George, 428
BPDN, Basis Pursuit Denoising, 440
BPSV, Backpropagation with Variable
Stepsize, 397
branch and bound, 24
- Breiman, Leo, 99, 132
Bremermann, Hans-Joachim, 121
Brent, Richard, 478
Brewer, Eric, 48
brillo, 178
Broyden, 529
Broyden, Charles George, 530
Bryson, Arthur E., 317
BTN, binary threshold neuron, 217
Buntine, Wray, 241
- C4.5, 99
C5.0, 99
cápsulas, 168, 635
célula amacrina, 173
célula bipolar, 173
célula ganglionar, 173, 174
célula horizontal, 173
código
del termómetro, 239
córtex, 163
visual, 169, 174
cabezas, 342
caja negra, 128, 156
cálculo
diferencial, 265
numérico, 266
camino aleatorio, 384
Campbell, Donald, 121
campo
electromagnético, 517
gravitatorio, 518
vectorial, 516
campo receptivo, 169, 615
Campoaomor, Ramón de, 179
cannabis, 166
capa
localmente conectada, 626
capa completamente conectada, 310
capa convolutiva, 614
conectividad, 616
profundidad, 615
stride, 617
zero padding, 617
capa de dropout, 464
capa de pooling, 627
stride, 630
ventana, 630
capa oculta, 205
capa softmax, 587
CapsNets, 635

- CAPTCHA, 43
 Carpenter, Gail, 206
 Carroll, Lewis, 122
 CART, 99, 132
 cascada, 342
 cascada de información, 130
 Casteljau, Paul de, 327
 Cauchy, Augustin Louis, 472
 Cauchy, Augustin-Louis, 316
 CBR, case-based reasoning, 107
 CBT, Cognitive-Behavioral Therapy, 57
 CD, 549
 Chameleon, 46
 Chen, Scott Shaobing, 440
 chimpancés, 57
 Chinook, 13
 Churchill, Winston, 59
 CI, Computational Intelligence, 30
 cicatrices de entrenamiento, 427
 ciencia de datos, 29
 científico vs. ingeniero, 161
 cine, 4, 18, 600
 CLARANS, 45
 Clark, Wesley, 221
 clasificación, 555
 - jerarquías de conceptos, 36
 - no balanceada, 37, 64
 - ontologías, 36
 clasificación de imágenes, 159, 653
 clasificador, 35
 - asociativo, 51
 - k-NN, 104
 - Naïve Bayes, 113
 - probabilístico, 36
 - top-k, 37
 clasificador cuadrático, 108
 clasificador de secuencias, 248
 clasificador lineal, 108
 clasificadores
 - asociativos, 102
 Clay Institute, 23
 CLIQUE, 49
 clustering, 43
 - basado en densidad, 47
 - en subespacios, 49, 135
 - espectral, 46
 - jerárquico, 46
 - por particiones, 44
 CNN, Convolutional Neural Network, 599
 aplicaciones, 651
 arquitectura, 637
 entrenamiento, 642
 seguridad, 650
 transferencia, 645
 visualización, 648
 Coase, Ronald, 428
 Coase, Ronald H., 48
 Coates, Adam, 444
 cobertura, 76
 cocaína, 166
 codificador-decodificador, 595
 coeficiente de silueta, 79
 coevolución, 122
 cognitron, 602
 colículo superior, 174
 Collins, Michael, 249
 Collobert, Ronan, 197
 colonias de hormigas, 120
 color, 178
 - constancia, 178
 - contraste, 178
 coloreado, 678
 columnas corticales, 167
 comecocos, 17
 complejidad, 77
 complejidad algorítmica, 22
 comportamiento irracional, 25
 composición, 148, 166
 - paralela, 148
 - secuencial, 149
 computación cuántica, 23
 computación evolutiva, 117, 120
 computación neuromórfica, 186
 concept drift, 392
 condensado de Hart, 105
 condicionamiento
 - clásico, 54
 - instrumental, 52
 - operante, 52
 conecta 4, 13
 conectividad de la red, 342
 conexiónistas, 125
 confianza, 49, 158
 conjunto de datos
 - de entrenamiento, 82
 - de prueba, 82
 - de validación, 83
 conjunto de entrenamiento, 60, 363
 conos, 173
 construcción Kesler, 245
 control PID, 20
 convergencia
 - aprendizaje online, 358
 - aprendizaje por lotes, 359
 - aprendizaje por mini lotes, 363
 - gradiente descendente, 366
 ConvNet, 599
 convolución, 606
 - 1x1, 616
 - dilatada, 667
 - mosaico (tiled), 627
 - zero padding, 608
 Cornsweet, Tom, 178
 Cortes, Corinna, 247
 costes, 63
 COTS HPC, 546
 Courville, Aaron, 145
 CPN, 206
 Cramer, Nichael, 125
 Cray-1, 153
 credibilidad, 157
 CRF, Conditional Random Field, 117
 cromosomas, 119
 Crowe, Russell, 371
 CRprop, 506
 CTRNN, continuous-time RNN, 188, 211
 cuadratura del círculo, 175
 cubo de Necker, 210
 CUDA, 153, 313
 CuBLAS, 314
 CuDNN, 314
 CURE, 46
 curriculum, 375
 curva ROC, 69
 CV, cross-validation, 84
 CVPR, 126
 Cybenko, George, 328, 329
 DAE, 453
 Dahl, George, 340
 damas, 13, 123
 Dartmouth, 1956, 10
 Darwin, Charles, 117, 456
 data mining, 29
 data science, 29
 datos
 - extracción de conocimiento, 29
 - procesamiento, 28
 - vs. información, 29
 DBN, 209

- DBSCAN, 47
DCGAN, Deep Convolutional GAN, 374
De Jong, Kenneth, 122
de Silva, Vin, 139
decorrelación, 365
deep learning, 125, 141, 260
DeepDream, 686
DeepFace, 675
DeepStack, 16, 58
Dekker, Theodorus, 478
delta-bar-delta, 400
deltap, 68
Dempster, Arthur, 112
DENCLUE, 47
dendrita, 185
dendrograma, 46
denoising autoencoder, 453
DenseNet, 666
derecho a recibir explicación, 157
DERprop, 505
descenso conjugado, 536
descenso por coordenadas, 549
descomposición en valores singulares, 138
detección de anomalías, 37, 64, 392
detección de fronteras, 608
detección de objetos, 669
DFS, 529
Dickson, William, 600
diferenciación automática, 317
diferentes estímulos, mismas activaciones, 175
dirección, 182
discretización, 43, 100
dispersión, 134
distancia, 44
dit, 560
divergencia, 517
divergencia Kullback-Leibler, 565
DLS, Damped Least-Squares, 524
DNC, Differentiable Neural Computer, 160
DNGO, 420
Domingos, Pedro, 122, 135
Donnelly, Peter, 140
Donoho, David, 440
dopamina, 166
double backprop, 375
Downing, Keith, 10
Downpour SGD, 543
dragones, 135
Dreyfus, Stuart E., 317
DropConnect, 468
dropout, 454, 462
boosting, 468
DropConnect, 468
fast dropout, 468
inverso, 465
regularización, 467
ruido gaussiano, 469
ruido multiplicativo, 466
softmax, 591
duda, 209
DuPont, 524
EANN, Evolving ANN, 424
early stopping, 84, 455, 473
EASGD, 547
ecuaciones de Maxwell, 517
ecualización de covarianzas, 365
edad, 676
edición de Wilson, 105
Edison, Thomas Alva, 600
EEBP, Equalized Error BackPropagation, 402
efecto
Abney, 178
apertura, 182
bandas de Mach, 177
Bezold-Brücke, 178
Casimir, 201
flash-drag, 182
flash-lag, 182
Fröhlich, 182
frontera de Cornsweet, 178
Hemlholz-Kohlrausch, 178
Hunt, 178
White, 177
efecto Baldwin, 119
efecto marco, 130
eficiencia, 77
Eigen, Manfred, 185
Einstein, Albert, 9, 428
elefantes, 428
eliminación de fondos, 676
ELM, 550
EM, Expectation Maximization, 47
EmBP, Emotional BackPropagation, 407
encoder-decoder, 595
enfermedad de costes, 3
enfriamiento simulado, 24
ensemble, 129, 430, 459
entrada neta, 192
entrenamiento, 28, 302, 323
basado en pistas, 375
complejidad, 330
con adversario, 369
criterios de parada, 303
datos, 363
early stopping, 304
estrategia recomendada, 325
funciones de activación, 343
hiperparámetros, 307, 412
inicialización, 304, 378
modos, 354
neurona lineal, 264
neurona oculta, 282
neurona sigmoidal, 278
online, 302, 356
orden de presentación, 306
pesos, 379
por capas, 643
por lotes, 303, 355
por minibatches, 360
pre-entrenamiento, 386
preprocesamiento, 305, 363
redes multicapa, 263
tasa de aprendizaje, 393
topología, 326
entrenamiento con adversario, 373
no supervisado, 374
semisupervisado, 373
supervisado, 373
entropía, 560
condicionada, 564
cruzada, 558, 566
de Gibbs, 561
relativa, 565
epigenética, 425
EQ, 253
equilibrio de Nash, 371
error, 62
accuracy ratio, 75
de resustitución, 81
MAE, 74
MAPE, 74
MASE, 75
MSE, 72
NMRSE, 73
RMSE, 73
sMAPE, 75

- SSE, 72
 error de Bayes, 106, 109
 escalabilidad, 41, 78
 espacio perceptual, 183
 espacios de versiones, 101
 estímulo-respuesta, 53
 estabilidad, 76
 estrategia de evolución, 123
 estratificación, 83
 Etzioni, Oren, 97
 evaluación
 cobertura, 76
 cohesión, 79
 complejidad, 77
 diversidad, 78
 eficiencia, 77
 escalabilidad, 78
 estabilidad, 76
 interconectividad, 79
 interpretabilidad, 77
 novedad, 78
 robustez, 76
 sensitivity, 66, 69
 separación, 79
 serendipia, 78
 silueta, 79
 specificity, 66
 evolución, 117
 Lamarckismo, 118
 ontogenética, 119
 exhaustividad, 65
 explicación, 158
 exploración vs. explotación, 121
 extracción de características, 137, 169
- F_β -score, 67
 F-score, 66
 Facebook
 Big Sur, 153
 DeepFace, 675
 M, 40
 FaceNet, 675
 fake, 676
 fall-out, 69
 falsas alarmas, 160
 falsos negativos, 61
 falsos positivos, 61
 Farley, Belmont, 221
 FBI, 675
 feature engineering, 140
 Fechner, Gustav, 183
- felicidad, 166
 fenómeno de Hughes, 109
 Fermi, Enrico, 428
 Feynman, Richard, 95
 fiabilidad, 160
 Fibonacci, 476
 Fidias, 476
 figura imposible, 209
 cubo, 210
 triángulo de Penrose, 210
 filogenética, 425
 filtrado colaborativo, 107
 filtro
 de histograma, 20
 de partículas, 20
 Kalman, 20
 filtro anti-spam, 111
 Fisher, Ronald, 121
 fitness, 117
 FitNets, 375
 Fletcher, Roger, 530
 Fliegende Blätter, 208
 flujo óptico, 183
 FN, False Negative, 61
 Fodor, Jerry, 33
 Fogel, David, 123
 Fogel, Lawrence, 120, 122
 FOIL, 101
 fórmula
 de Sherman-Morrison, 528
 FP, False Positive, 61
 FPR, False Positive Rate, 69
 franja dorsal, 174
 franja ventral, 174
 Fraser, Alex, 121
 frecuencia de muestreo, 600
 frecuentista, 156
 Freitas, Nando de, 539
 Freud, Sigmund, 166
 Freund, Yoav, 132, 246
 Friedman, Milton, 428
 Fritz, 123
 FTRL, 514
 Fujiki, Cory, 125
 Fukushima, Kuhiniko, 602
 Funahashi, Ken-ichi, 328
- función
 circular, 200
 de Borel, 329
 de error, 261
 de Heaviside, 196
- de pérdida, 261
 escalón, 196, 218
 gudermanniana, 200
 hiperbólica, 200
 logística, 197
 sigmoidal bipolar, 199
 trascendental, 200
 umbral, 196
 función de Ackley, 485
 función de activación, 191, 194, 343
 con saturación, 196
 cos, 353
 escalón, 196
 hard tanh, 353
 leaky ReLU, 202, 351
 lineal, 195
 maxout, 203, 351
 PReLU, 202, 351
 RBF, 203, 352
 ReLU, 201, 349
 sigmoidal, 197, 343
 softplus, 201, 353
 valor absoluto, 202, 350
 función de transferencia, 194
- Gallant, Ronald, 328
 Gallant, Stephen, 240, 320
 Galton, Francis, 130
 GAN, Generative Adversarial Network, 369, 635, 676
 Gauss, Carl Friedrich, 522
 GECCO, 118
 gen, 119
 generalización, 147
 Gentile, Claudio, 248
 geometría, 179
 hiperbólica, 201
 longitud, 179
 occlusión, 180
 parallelismo, 180
 superficie, 181
 Gibbs
 desigualdad, 565
 entropía, 561
 GIGO, 26
 Glorot, Xavier, 340, 350, 382
 glutamato, 173
 GMM, Gaussian Mixture Model, 47, 578
 Go, 15
 AlphaGo, 15

- GOFAI, 97
gold standard, 63
Goldfarb, Donald, 530
Good, Irving John, 560
Goodfellow, Ian, 145, 370
Google, 38, 660
 - AlphaGo, 15
 - Assistant, 40
 - AutoML, 422
 - DeepDream, 686
 - DistBelief, 543, 547
 - Knowledge Graph, 40
 - PageRank, 38
 - RankBrain, 39
 - Street View, 658
 - TensorFlow, 544
 - Translate, 392
 - Translator, 33
 - YouTube, 658
GoogLeNet, 660
GPS, General Problem Solver, 11
GPS, Global Positioning System, 3, 20
GPU, 152, 193, 314
grabados, 613
gradient boosting, 133
gradiente
 - desaparición, 293, 399
 - descendente, 266
 - evanescente, 293
 - explosión, 399
 - implementación, 309
 - natural, 596
gradiente descendente
 - asíncrono, 542
 - convergencia, 366
 - estocástico, 487
 - proyectado, 446
gradientes biconjugados, 535
gradientes conjugados, 534
grandmother cell, 170
Grisham, John, 107
Grossberg, Stephen, 206, 258
Grover, Lov, 23
GRprop, 506
Gudermann, Christoph, 200
gusto, 175

Hare, Brian, 57
Hart, Peter, 12, 104
hartley, 560
Hartley, Ralph, 560

Hassabis, Demis, 58
Haugeland, John, 97
Hawkins, Jeff, 168
Hayek, Friedrich, 222
Heaviside, Oliver, 218
Hebb, Donald, 163, 217, 319
Hecht-Nielsen, Robert, 206, 328
Heckerman, David, 111, 116
Heisenberg, Werner, 96
Hering, Ewald, 181
Hesse, Ludwig Otto, 516
hessiano, 516
Hestenes, Magnus, 535
heterocedasticidad, 578
heurísticas, 22
HF, Hessian-free, 522
Hicklin, Joseph, 125
Hierarchical Temporal Memory, 168
higrómetro, 200
Hillis, Danny, 122
Hinton, Geoffrey, 52, 120, 127, 128, 144, 151, 153, 156, 168, 261, 318, 340, 352, 507
hipótesis de la Reina Roja, 122
hipótesis de la variedad, 337, 340
hiper-naranjas, 135
hiperparámetro
 - inicialización, 384
 - regularización, 435
hiperparámetros, 83, 128, 412
 - ajuste automático, 415
 - ajuste manual, 412
 - búsqueda aleatoria, 417
 - búsqueda inteligente, 418
 - búsqueda sistemática, 416
 - optimización basada en modelos, 418
 - optimización bayesiana, 418
hipocampo, 163, 168
Ho, Yu-Chi, 317
Hodgkin, Alan Lloyd, 188
Hoff, Ted, 238
Hofstadter, Douglas, 104
HOG, 670
Hogwild, 543
Holland, John, 120–122
Hollywood, 4, 17
homúnculo, 172
Hopfield, John, 208
horizonte, 180
Hornik, Kurt, 328
Horvitz, Eric, 25

Howard, Ron, 371
HTM, 168, 636
Hubel, David, 169, 601
Hume, David, 30
Huxley, Andrew Fielding, 188
Hyndman, Rob, 75
Hypergrad, 421
Hyperopt, 420

I.A., 3
IBM
 - Deep Blue, 14, 123
 - Quest, 49, 99
 - Watson, 21, 39, 107
ICML, 118, 126
ID3, 99
ILP, Inductive Logic Programming, 101
ILSVCR, 127
ILSVRC, 659
ilusión
 - caras/copa, 209
 - conejo/pato, 208
 - contraste simultáneo, 177
 - de Hering, 181
 - de Müller-Lyer, 180
 - de Orbison, 181
 - de Poggendorff, 181
 - de Wundt, 181
 - de Zöllner, 181
 - horizontal-vertical, 180
 - jarrón de Rubin, 209
 - Necker, 210
ilusión visual, 209
ilusiones perceptuales, 176
imágenes
 - afilado, 612
 - descripción textual, 594
 - detección de fronteras, 608
 - grabado, 613
 - suavizado, 610
ImageNet, 393, 659
implementación, 294
 - depuración, 308
 - GPU, 313
 - modular, 310
imputación de valores, 100
Inception, 660
inceptionismo, 686
incidente del lago Tahoe, 118
indexación

- Fukunaga-Narendra, 105
índice
de Youden, 68
inducción de reglas, 101
información
de Fisher, 566, 598
mutua, 564, 566
teoría, 559
informe ALPAC, 12
informe Lighthill, 12
ingeniería de características, 140
inicialización, 378, 490
capa de salida, 386
dispersa, 384
fan-in, 381
fan-out, 383
hiperparámetro, 384
LSTM, 386
Nguyen y Widrow, 381
pesos, 379
ReLU, 383
sesgos, 385
softmax, 589
unidades de control, 386
Xavier, 382
inspiración biológica, 160
Intel
MKL, 314
Xeon Phi, 314
inteligencia
definición, 8
Inteligencia Artificial, 3
computación antrópica, 9
definición, 8
problemas, 10
racionalidad computacional, 9
inteligencia computacional, 30
interpretabilidad, 41, 77, 128, 157
intuición, 158
invarianza
por entrenamiento, 642
por estructura, 642
invierno de la I.A., 12, 256
Ioffe, Sergey, 410
iRprop, 505
Isomap, 139
- J, 68
J4.8, 99
Jackel, Larry, 151
Jacobi, Carl Gustav Jacob, 515
jacobiano, 515
James, Williams, 121
Jarrett, Kevin, 350
Jevons, William Stanley, 30
Jordan, Michael, 111, 128, 140
Judd, J. Stephen, 330
jurisprudencia, 107
- k-CV, 84
k-means, 45, 112
k-medoids, 45
k-modes, 45
k-NN, 104
Kaggle, 130, 424
Kaplan, Jerry, 9
Karpathy, Andrej, 645
Kasparov, Gary, 14
Kavukcuoglu, Koray, 350
KD, Knowledge Distillation, 375
KDD, Knowledge Discovery in Databases, 29
Kelley, Henry J., 317
Kepler, Johannes, 476
Keynes, John Maynard, 59
Khronos Group, 313
KKT, 549
Klein, Dan, 9
Kleinberg, Jon, 48
Kohonen, Teuvo, 107, 128, 206
Kolmogorov, Andrei, 328
Koza, John, 118, 120, 125
Krauth, Werner, 242
Krizhevsky, Alex, 127, 145, 659
Kuffler, Stephen, 601
Kullback, Solomon, 565
- L'Hôpital, Guillaume de, 315
L-BFGS, 531
Sandblaster, 547
L-BFGS-B, 533
L1, 438
L2, 436
lógica
proposicional, 216
límite de Landauer, 563
label smoothing, 590
Laboratorios Bell, 151
Lagrange, Joseph Louis, 445
Lagrange, Joseph-Louis, 315
Laird, Nan, 112
Lamarck, Jean Baptiste, 119
Lambert, Johann Heinrich, 200
Landauer, Rolf, 562
Langford, John, 139
Laplace, suavizado, 114
laplaciano, 516
LASSO, 439
layer-wise pretraining, 643
lazy learners, 104
LDA, Latent Dirichlet Allocation, 140
lectura de labios, 33
LeCun, Yann, 125, 128, 144, 151, 318, 324, 350, 508, 604
LEGO, 148, 150
Leibler, Richard, 565
Leibniz, Gottfried Wilhelm, 315
LeNet, 604
Lettvin, Jerry, 170
Levenberg, Kenneth, 524
ley
de Hiram, 59
ley de Hiram, 157
ley de Say, 8
leyes de la robótica, 6
LFW, 675
LIDAR, 19
lift, 62
likelihood, 473
limitador estricto, 196
Linnainmaa, Seppo, 317
listas de decisión, 101
Littlestone, Nick, 236
LM-BFGS, 531
LMS, 238
lobos, 57
localización, 169
Logic Theorist, 11
logit, 280, 571
Lømo, Terje, 319
Lorente de Nò, Rafael, 167
Lovecraft, H.P., 255
LSTM
inicialización, 386
LSUV, Layer-Sequential Unit-Variance, 384
LTM, 165
LTP, Long-Term Potentiation, 319
LTU, linear threshold unit, 217
Ludd, Ned, 7
ludditas, 7
luminancia, 176
contexto, 176

- ilusión de Munker, 177
interpretación en 3D, 177
- Luna, 180
- LVQ, 206
- LVQ, linear vector quantization, 107, 128
- máquina de Boltzmann, 209
restringida, 209
- máquina lineal, 245
- máquinas de vectores de soporte, 108
- método de los multiplicadores de Lagrange, 445
- métodos de agrupamiento, 43
basados en densidad, 47
en subespacios, 49, 135
jerárquicos, 46
por particiones, 44
- métodos de condensado, 105
- métodos de edición, 105
- mínimo local, 483
- Mach, Ernst, 178
- machine learning, 27
- MAE, Mean Absolute Error, 74
- maldición de la dimensionalidad, 134, 340
- manifold, 135, 337
- MAPE, Mean Absolute Percentage Error, 74
- Mark I Perceptron, 224
- Markowitz, Harry, 25
- Marquardt, Donald, 524
- Marr, David, 652
- Martens, James, 340
- MASE, Mean Absolute Scaled Error, 75
- materia
blanca, 187
gris, 187
- matriz
hessiana, 516
jacobiana, 515
- matriz de confusión, 60
- matriz ortogonal aleatoria, 384
- Max-Over, 242
- MAXNET, 206
- maxout, 203, 351, 637
- Maxwell, James Clerk, 517
- Mays, Crowell Hugh, 237
- MCC, Matthews correlation coefficient, 68
- McCarthy, John, 10
- McClelland, James, 148, 256
- McCulloch, Warren, 215
- MDN, Mixture Density Network, 578
- mecanismos de atención, 594
- memoria, 596
- Mendel, Gregor, 119
- meta-aprendizaje, 131
- metaheurísticas, 24, 117
- metamodelo, 131
- método
de Gauss-Newton, 522
de Levenberg–Marquardt, 524
de Newton, 520
SFN, 522
truncado, 522
- quasi-Newton, 527
- metodología, 324
prototipado, 325
- metodología STAR, 101
- métodos de continuación, 550
- Mézard, Marc, 242, 321
- Michalski, Ryszard, 97
- Microsoft, 662
Adam, 545
Cortana, 40
DirectCompute, 313
Office, 116
Research, 145, 367
traducción simultánea, 21
Windows, 116
- mielina, 187
- Miller, George, 165
- Milner, Brenda, 168
- Min-Over, 242
- minería de datos, 29
- Minsky, Marvin, 10, 12, 222, 255
- mismos estímulos, diferentes activaciones, 175
- Mitchell, Tom, 97
- mitocondrias, 119
- MLE, maximum likelihood estimation, 575
- MLP, multilayer perceptron, 256, 259
- MMX, 152
- MNIST, 244, 247, 367, 467, 469, 540, 657
- MobileNet, 668
- modelo, 185, 428
de Hodgkin-Huxley, 188
de Izhikevich, 188, 211
- de McCulloch y Pitts, 215
espacial, 187
neurociencia, 210
temporal, 187
- modelo BTL, 581
- modelo conexionista, 318
- modelo de clasificación, 35
- modelo del cubito de hielo, 169
- modelos analógicos, 102
- modelos de neurona artificial, 187
- modelos de redes neuronales artificiales, 204
- modelos probabilísticos, 110
Naïve Bayes, 113
redes bayesianas, 114
redes de Markov, 116, 117
- modularidad, Q, 47
- momento, 301, 404, 492
de Nesterov, 406, 496
NAdam, 513
RMSprop, 508
tradicional, 494
velocidad terminal, 493
- Montúfar, Guido, 340
- morfogénesis, 425
- Morgenstern, Oskar, 13
- mosaico, 627
- motif discovery, 51
- Mountcastle, Vernon, 601
- movimiento, 169, 181
- MPI, 547
- MPT, Modern Portfolio Theory, 25
- MRF, Markov Random Field, 116
- MRR, Mean Reciprocal Rank, 68
- MSE, Mean Squared Error, 72
- muestreo, 43
- Muggleton, Stephen, 97
- Multiedit, 105
- n-CV, 85
- núcleo supraquiasmático, 174
- número de condición, 366
- Naïve Bayes, 113
- Nadal, Jean-Pierre, 321
- NAdam, 513
- NAE, 454
- NAG, Nesterov Accelerated Gradient, 406, 496
- Nair, Vinod, 352
- Nash, John Forbes, 371
- nat, 560

- nature vs. nurture, 162
 navaja de Occam, 99, 428
 Necker, Louis Albert, 210
 Necronomicon, 255
 neocognitron, 602
 nervio óptico, 174
 Netflix, 33
 competición, 130
 neurociencia computacional, 161, 186
 neurodo, 191
 neuroevolución, 424
 codificación directa, 425
 codificación indirecta, 425
 neurona, 191
 binaria, 194
 bipolar, 194
 con memoria, 192
 de McCulloch y Pitts, 215
 estocástica, 193
 no lineal, 192
 neurona abuela, 170
 neurotransmisor, 166
 Newell, Allen, 10
 Newman, Mark, 47
 Newton, Isaac, 522
 Ng, Andrew, 140, 324, 444
 Nguyen, Derrick, 381
 nicotina, 166
 Nilsson, Nils, 12
 Nim, juego de, 13
 NIPS, 126
 NMF, 549
 noisy autoencoder, 454
 normalización, 305, 363
 normalización por lotes, 409, 455
 NoSQL, 48
 Nowlan, Steven, 120
 NRMSE, Normalized RMSE, 73
 NTM, Neural Turing Machine, 160, 596
 nubes de partículas, 24
 NVIDIA, 152
 DGX-1, 153
 GTX, 153
 Nvidia
 CuBLAS, 314
 CUDA, 313
 Nyquist, Harry, 171
- O-LBFGS, 533
 oído, 171
 Ockham, William, 99
- OCR, 21, 452
 olfato, 175
 OLL, Optimization Layer by Layer, 409
 OMP, Orthogonal Matching Pursuit, 444
 one-hot, 243
 one-shot learning, 392
 ontogénesis, 425
 OpenCL, 313
 clBLAS, 314
 operador
 de Laplace, 516
 opiáceos, 166
 opsina, 173
 OPTICS, 47
 optimalidad acotada, 25
 optimización, 471
 óptimo local, 483
 búsqueda de la sección áurea, 476
 búsqueda lineal, 472
 backtracking, 532
 búsqueda ternaria, 475
 con restricciones, 208, 445
 convexa, 484
 de segundo orden, 514
 interpolación parabólica inversa, 477
 método de Brent, 478
 método de Nelder-Mead, 480
 método de Powell, 481
 mínimo local, 483
 no convexa, 484
 regiones de confianza, 472
 uso de derivadas, 479
 Orbison, William, 181
 organización jerárquica, 166
 orientación, 169, 180
 OSS, 533
 Othello, 15
 outliers, 42
 OverFeat, 670
 overfitting, 82, 129
 OWO-HWO, 550
- P vs. NP, 23
 pájaro, 161
 péndulo invertido, 201
 pérdida 0-1, 62
 Pac-Man, 17
 palomas, 53
 PAM, 45
- Papadimitriou, Christos, 122
 Papert, Seymour, 12, 255
 parámetros
 compartidos, 448
 compresión, 447
 PARA, 223
 paradoja
 de los falsos positivos, 64
 Ingeniería del Conocimiento, 59
 parallelización, 540
 parallelización de algoritmos, 193
 Parker, David, 317
 patrón oro, 63
 patrones frecuentes, 50
 Pavlov, Ivan, 54
 PCA, Principal Component Analysis, 138, 364
 decorrelación, 365
 reducción de dimensionalidad, 365
 PDP Group, 148
 PDP, Parallel Distributed Processing, 256, 318
 PE, processing element, 187, 191
 Pearl, Judea, 111
 Pearson, Karl, 138
 películas, 4, 18
 percepción, 170
 frío y calor, 220
 interacciones, 183
 perceptrón, 223
 Σ - Π , 235
 α , 235
 adaptación incremental, 237
 AdaTron, 242
 ALMA, 248
 aproximador universal, 235
 arquitectura, 224, 226
 bolsillo, 240
 convergencia, 231
 corrección absoluta, 237
 entrenamiento, 226
 estructurado, 248
 gamba, 256
 incremental, 250
 interpretación geométrica, 229
 limitaciones, 251
 Max-Over, 242
 Min-Over, 242
 multiclase, 243
 promedio, 245
 relajación modificada, 238

- sigmoidal, 239
- tasa de aprendizaje, 236
- variantes, 235
- zona muerta, 237
- perros, 53, 57, 164
- perspectiva, 179
- peso, 186
- peso sináptico, 192
- Picasso, Pablo, 31
- PID, 407
- Pitts, Walter, 215
- plasticidad, 163
 - estructural, 164
 - intrínseca, 164
 - sináptica, 164
- poda, 426
 - poda alfa-beta, 13, 14
- Poggendorff, Johann Christian, 181
- poker, 16
- Polyak, Boris, 492
- potencial, 518
- pre-entrenamiento, 386
 - no supervisado, 388
 - supervisado, 387
- pre-entrenamiento no supervisado, 43
- precipicio, 490
- precisión, 41
- precisión, accuracy, 61
- precisión, precision, 65
- PRelu, 202, 351
- premios Humie, 125
- preprocesamiento
 - ampliación, 367
- preprocesamiento de datos, 42, 363
 - decorrelación, 365
 - ecualización de covarianzas, 365
 - eliminación de medias, 364
 - escalado, 364
 - normalización, 363
 - PCA, 364
 - reducción de dimensionalidad, 365
- principio de perturbación mínima, 397
- Pritchard, Jonathan, 140
- probabilidad, 156
- problema óptico inverso, 175
- problema de la identificabilidad, 484
- problemas del milenio, 23
- problemas NP-completos, 23
- problemas NP-difíciles, 23
- procesamiento de imágenes, 676
 - convolución, 606
- PROCLUS, 49
- productividad, 3
- profundidad de la red, 336
- programación dinámica, 20
- programación evolutiva, 122
- programación genética, 124
- promediado de Polyak, 490
- promediado elástico, 546
- PRONG, PROjected Natural Gradient, 411
- propagación tangencial, 374
- propiedades asintóticas
 - consistencia, 93
 - eficiencia, 93
- proteína G, 563
- pseudoinversa de Moore-Penrose, 438
- psicofísica, 183
- PUBLIC, 101
- punto de silla, 485
- pupila, 174
- Purves, Dale, 175
- PVM, 547
- QA, Question Answering, 39
- QRprop, 505
- Quickprop, 402, 499
- Quinlan, Ross, 97
- R-CNN, 670
 - Fast, 671
 - Faster, 671
 - Mask, 672
 - R-FCN, 672
- R-FCN, 672
- RainForest, 101
- Ramón y Cajal, Santiago, 167
- ramificación y poda, 24
- RAND Corporation, 11
- random forest, 132
- random walk, 384
- ranking, 36, 65
- Ranzato, Marc'Aurelio, 350
- Raphael, Bertram, 12
- razonamiento abductivo, 103
- razonamiento analógico, 103
- razonamiento basado en casos, 107
- RBF, 203
 - gaussiana, 204
 - normalizada, 594
- RBM, 209
- recall, 65
- receptor $\beta 2$, 563
- Rechenberg, Ingo, 120, 123
- reconocimiento de formas, 103
- reconocimiento de patrones, 32, 103
- reconocimiento de voz, 12, 452, 595
- reconocimiento facial, 673
- recorte del gradiente, 399, 490
- recuperación de información, 38
- red
 - ART, 206
 - competitiva, 206
 - completamente conectada, 260
 - con múltiples cabezas, 342
 - con saltos, 342
 - contrast-enhancement, 206
 - de contra-propagación, 206
 - de Hamming, 206
 - de Hopfield, 208
 - de mejora de contraste, 206
 - en cascada, 342
 - feed-forward, 204, 259
 - LVQ, 206
 - MAXNET, 206
 - multicapa, 205, 259
 - pattern-completion, 208
 - profunda, 205
 - recurrente, 207
 - siamesa, 674
 - simple, 204
 - simulación, 210
 - SOM, 206
 - sombrero mexicano, 206
- red neuronal, 125
- red neuronal multicapa, 127
- redes
 - estudiantes y profesores, 375
 - redes bayesianas, 114
 - redes con adversario
 - ejemplos, 372
 - ejemplos virtuales, 373
 - entrenamiento, 373
 - GAN, 370
 - redes de Markov, 116
 - redes elásticas, 441
 - reducción de dimensionalidad, 134, 365
 - extracción, 137
 - selección, 136
 - reducción de la dimensionalidad, 43
 - regla
 - de la cadena, 315
 - de Widrow-Hoff, 238

- delta, 238, 270, 272
 - ejemplo, 268
 - delta generalizada, 239
 - LMS, 238
- regla delta generalizada, 290
- reglas de asociación, 49
- regresión, 72, 555
 - lineal, 578
 - logística, 578
 - logística multinomial, 579
 - regularizada, 436
 - softmax, 593
- regularización, 129, 156, 432
 - de Tikhonov, 437
 - dropout, 462
 - early stopping, 455
 - ensemble, 459
 - función de coste, 434
 - L1, 438
 - L2, 436
 - parámetros, 445
 - redes elásticas, 441
 - ridge, 437
 - ruido, 450
 - softmax, 590
 - weight decay, 436
- relación de aspecto, 600
- ReLU, 201
 - inicialización, 383, 386
- rendimiento
 - análisis, 326
 - métricas, 325
- representación
 - distribuida, 341
 - no distribuida, 341
- representación distribuida, 148
- representation learning, 142
- reproducción sexual, 122
- ResNet, 662
- resolución, 600
- responsabilidad algorítmica, 157
- retina, 169, 172
- Reversi, 15
- Rich, Elaine, 10
- Riedl, Mark, 679
- riesgo, 473
 - empírico, 473
- Rivest, Ronald Linn, 331
- RLS, Recursive Least Squares, 438
- RMSE, Root Mean Squared Error, 73
- RMSprop, 506
- momento de Nesterov, 508
- RNN, Recurrent Neural Network
 - vs. TDNN, 605
- robótica, 34
 - leyes, 6
- robustez, 76
- ROC, curva, 69
- Rochester, Nathaniel, 10
- ROCK, 46
- Romer, Paul, 96
- Rosenblatt, Frank, 151, 221, 222
- rotacional, 517
- Rprop, 503
 - CRprop, 506
 - DERprop, 505
 - iRprop, 505
 - QRprop, 505
 - SARprop, 505
 - SASS, 505
- Rprop-GRprop, 506
- rubber banding, 17
- Rubin, Donald, 112
- Rubin, Edgar, 209
- ruido, 450
 - sobre las capas ocultas, 454, 466
 - sobre las entradas, 451
 - sobre los pesos, 453
- Rumelhart, David, 256, 261, 318
- Ruppert, David, 492
- Russell, Bertrand, 11, 79
- Russell, Stuart, 25
- S3, 152
- símbolo, 97
- sabiduría de las multitudes, 130
- Sagan, Carl, 33
- Salakhutdinov, Ruslan, 153
- saltos, 342
- Samad, Tariq, 300
- Samuel, Arthur, 10, 13, 27
- Sandblaster L-BFGS, 547
- SARprop, 505
- SASS, 505
- saturación, 178, 343
- Say, Jean-Baptiste, 8
- SCAWI, Statistically-Controlled Activation Weight Initialization, 385
- Schapire, Robert, 132, 246
- Schultz, Wolfram, 55
- Schwefel, Hans-Paul, 123
- SCN, 174
- señal, 599
 - continua, 599
 - discreta, 599
- señales de tráfico, 159
- sector cuaternario, 3
- segmentación de imágenes, 669
- segmentación semántica, 676
- seguridad, 372, 650
- selección
 - elitista, 121
 - estocástica, 121
 - extintiva, 124
 - natural, 117
 - probabilística, 121
- selección de características, 136
 - filtro, 136
 - incrustada, 137
 - wrapper, 136
- selección de instancias, 43, 136
- selectividad, 169
- sensores, 171
- sentidos, 170
- serendipia, 78
- series temporales, 72
- sesgo, 90, 129, 191, 192, 431
 - sesgo cognitivo, 25, 130
 - sesgo psicológico, 130
- SFN, 522
- SGD, 487
 - asíncrono, 542
 - Downpour, 543
 - Hogwild, 543
 - promediado elástico, 547
 - tolerante a retardos, 546
- SGEMM, 314
- Shanno, David, 530
- Shannon, Claude, 10, 171, 560
- Shaw, Cliff, 11
- siamesa, red, 674
- sigmoide, 197
 - binaria, 197
 - bipolar, 199
 - gd, 200
 - logística, 197
 - tanh, 199
- significado, 97
- significante, 97
- signo, 97
- Sima, Jiri, 331
- Simard, Patrice, 367
- SIMD, 193

- simetría en el espacio de pesos, 484
 similitud, 44
 similitud, 103
 Simon, Herbert, 10, 28
 simulación
 tiempo discreto, 219
 sistemas clasificadores, 122
 tipo Michigan, 122
 tipo Pittsburgh, 122
 sistemas de recomendación, 40, 107, 159
 sistemas expertos, 13, 59, 97, 142
 sistemas inmunes artificiales, 120
 Skinner, B.F., 53
 SLAM, 20
 SMAC, 420
 sMAPE, Symmetric MAPE, 75
 Smith, Stephen, 122
 SNARC, 222
 SNN, 47
 Snow, Jon, 104
 sobreaprendizaje, 82, 154, 428
 prevención, 155
 Socher, Richard, 58
 soft computing, 30
 softmax, 555
 capas ocultas, 593
 dropout, 591
 estabilidad numérica, 581
 función, 569
 función de coste, 583
 gradiente, 586
 implementación, 587
 inicialización, 589
 jerárquico, 592
 regresión, 593
 regularización, 590
 smoothing, 590
 weight decay, 590
 softplus, 201
 softsign, 344
 Sol, 180
 SOM, Self-Organizing Map, 206
 soma, 185
 Sony
 AIBO, 58
 soporte, 49
 Spearmint, 420
 spike, 185
 spiking model, 188
 Izhikevich, 211
 SRM, 210
 Sprecher, David A., 328
 SR1, 529
 SRM, Spike Response Model, 188, 210
 SSD, 673
 SSE, 152
 SSE, Sum of Squared Errors, 72
 stacking, 133
 Stanford Research Institute, 12
 Stephens, Matthew, 140
 Stiefel, Eduard, 535
 Stinchcombe, Maxwell, 328
 STM, 165
 STN, Spatial Transformer Network, 668
 Stone, Marshall Harvey, 327
 Stratego, 16
 suavizado de etiquetas, 590
 suavizado de imágenes, 610
 suavizado de Laplace, 114
 subsimbólicos, 125
 super-resolución, 678
 supercomputador vectorial, 153
 SuperSAB, 401
 Sutskever, Ilya, 127, 145
 Sutskever, Ylya, 340
 SVD, Singular Value Decomposition, 138
 SVHN, 658
 SVM, Support Vector Machine, 108, 151, 247
 swarm intelligence, 120
 Szegedy, Christian, 410
 tálamo, 169, 174
 tacto, 172
 tamaño de la red, 334
 tangent prop, 374
 tangente hiperbólica, 199
 estricta, 197
 Target, 157
 tasa de aprendizaje, 236, 393, 497
 aproximación estocástica, 395
 búsqueda y convergencia, 395, 498
 conductor audaz, 498
 descenso lineal, 396
 disminución lineal, 498
 heurísticas, 394
 local, 400, 499
 momento, 404
 tasa de error, 62
 taxista, 4, 163
 Tay, Thinking about You, 28
 Taylor, W.K., 221
 TD-learning, 54, 165
 TDIDT, 99
 TDNN, Time-Delay Neural Network, 605
 vs. RNN, 605
 televisión, 600
 Tenenbaum, Joshua, 139
 tensor
 3D, 626
 4D, 626
 6D, 626
 TensorFlow, 544
 teoría, 185
 teoría de control, 316
 teoría de juegos, 13
 Teoría de la Información, 559
 teoría moderna de selección de cartera, 25
 teorema
 CAP, 48
 de aproximación universal, 327
 de imposibilidad, 48
 de Stone-Weierstrass, 327
 no free lunch, 41, 329
 test de Lovelace, 679
 test de Turing, 11, 679
 Thorndike, Edward, 53, 216
 Tibshirani, Robert, 439
 Tielemans, Tijmen, 507
 Tikhonov, Andrey, 437
 TN, True Negative, 61
 TNR, True Negative Rate, 66
 tono, 178
 topología, 326
 aproximador universal, 327
 con múltiples cabezas, 342
 con saltos, 342
 conectividad de la red, 342
 en cascada, 342
 profundidad de la red, 336
 tamaño de la red, 334
 TP, True Positive, 61
 TPR, True Positive Rate, 66, 68
 traducción automática, 11, 21, 33, 595
 transfer learning, 82
 transformación de características, 137
 transputer, 547
 trigonometría

- esférica, 199
- triplet loss, 675
- TRPO, 597, 598
- TSP, 208
- TTBP, Three Term BackPropagation, 407
- TTS, 21
- Tukey, John, 560
- Turing, Alan, 11, 221, 425, 560
- twinning, 86
- umbral de activación, 192
- underfitting, 129
- Utgoff, Paul, 241
- Uttley, Albert M., 221
- V1, 174, 637
- V2, 174
- VAE, Variational Autoencoder, 635
- validación cruzada, 84
- Vapnik, Vladimir, 104, 151, 247
- varianza, 90, 129, 431
- variedad, 135, 337
- vecinos más cercanos, 104
- vehículos autónomos, 4, 19
- velocidad, 181
- verosimilitud, 473
- VGG, 662
- videojuegos, 16, 426
 - carreras, 17
 - comecocos, 17
- Viola-Jones, detector, 669
- visión, 171
 - contraste, 173
 - desenfoque, 173
 - movimientos sacádicos, 172
 - resolución, 171
- visión artificial, 19
- visión estereoscópica, 169
- VNNIW, 152
- Voltaire, 24
- von Neumann, John, 13, 428
- Voyager, 33
- VR, variable reinforcement, 53
- vSGD, 508
 - vSGD-fd, 509
 - vSGD-g, 509
 - vSGD-l, 509
- VTLP, Vocal Tract Length Perturbation, 452
- vulnerabilidad, 372
- Wan, Eric, 318
- Weierstrass, Karl, 200, 327
- weight decay, 436
 - softmax, 590
- weight noise, 453
- weight sharing, 448
 - soft, 450
- Weinberg, Gerald, 9
- Wendemuth, Andreas, 242
- Werbos, Paul, 261, 317
- White, Halbert, 328
- White, Michael, 177
- Whitehead, Alfred, 11
- Widrow, Bernard, 238, 239, 381
- Wiesel, Torsten, 169, 601
- Williams, Ronald, 261, 318
- Wilson, Stewart, 122
- Winnow, 236
- wisdom of crowds, 130
- Wittgenstein, Ludwig, 208
- Wolfram Alpha, 40
- Wolpert, David, 41, 134, 333
- word2vec, 393
- Wundt, Wilhelm, 181
- Xception, 667
- XGBoost, 133
- XNOR, 253
- XOR, 252
 - perceptrón, 257
 - ReLU, 257
- YOLO, 672
- YouTube, 658
- z-score, 305, 363, 364
- Zöllner, Johann Karl Friedrich, 181
- Zeiler, Matthew, 502
- zero-data learning, 392
- zero-shot learning, 392
- ZFNet, 660
- zorros, 57